Projet SYNCHRONE : les formats communs des langages synchrones

Jean-Pierre Paris, Gérard Berry, Frédéric Mignard, Philippe Couronné, Paul Caspi, Nicolas Halbwachs, Yves Sorel, Albert Benveniste, Thierry Gautier, Paul Le Guernic, et al.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

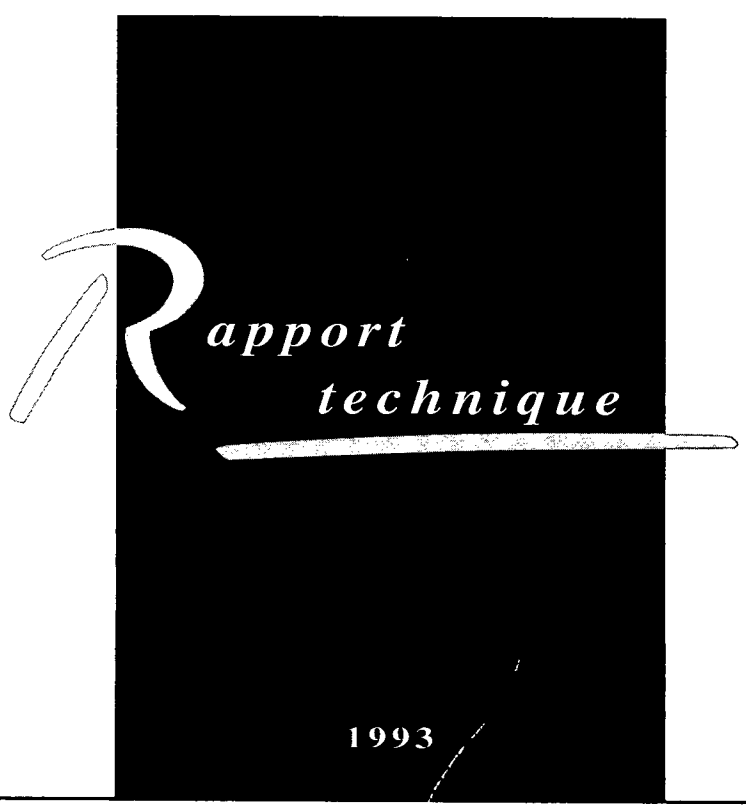
Projet SYNCHRONE
Les formats communs des langages synchrones

Jean-Pierre PARIS - Gérard BERRY - Frédéric MIGNARD
Philippe COURONNÉ - Paul CASPI - Nicholas HALBWACHS
Yves SOREL - Albert BENVENISTE - Thierry GAUTIER
Paul LE GUERNIC - François DUPONT - Claude LE MAIRE

N° 157
Juin 1993

PROGRAMME 2

Calcul symbolique,
programmation
et génie logiciel



Collaboration "CAO-Automatique" (C2A)
PRC "Communication, Coopération, Concurrence" (C^3)
GRECO Automatique
Cercle pour les Projets Innovants en Informatique (CP2I)

PROJET SYNCHRONE

Les formats communs des langages synchrones

Version 0

Ce document a été rédigé par les auteurs ci-dessous, cités selon leur organisme d'appartenance.

Jean-Pierre PARIS	CISI Ingénierie / Agence Provence-Est Les Cardoulines - Bât. B1 – Route des Dolines Sophia Antipolis – 06560 Valbonne
Gérard BERRY, Frédéric MIGNARD	École des Mines de Paris / C.M.A. - INRIA B.P. 207 – 06904 Sophia-Antipolis Cedex
Philippe COURONNÉ	ILOG S.A. – 2, Av. Galiéni – 94253 Gentilly
Paul CASPI, Nicholas HALBWACHS	IMAG – B.P. 53 – 38041 Grenoble Cedex
Yves SOREL	INRIA – Domaine de Voluceau Rocquencourt – B.P. 105 – 78150 Le Chesnay
Albert BENVENISTE, Thierry GAUTIER, Paul LE GUERNIC	IRISA/INRIA – Campus de Beaulieu 35042 Rennes Cedex
François DUPONT	TNI – ZI du Vernis – 29608 Brest
Claude LE MAIRE	VERILOG – Centre d'Études Rhône-Alpes ZAC du Pré Millet – 38330 Montbonnot

Les formats communs des langages synchrones ont été élaborés par les auteurs du document sur la base de travaux de recherche conduits au CMA, à l'IMAG et à l'IRISA. Les auteurs remercient les participants à ces travaux pour leurs diverses contributions.

La maîtrise de rédaction a été exercée, dans une première phase par Nicholas HALBWACHS, puis dans la phase finale par Thierry GAUTIER et Paul LE GUERNIC.

Collaboration "CAO-Automatique" (C2A)
PRC "Communication, Coopération, Concurrence" (C^3)
GRECO Automatique
Cercle pour les Projets Innovants en Informatique (CP2I)

PROJET SYNCHRONE

Les formats communs des langages synchrones

Version 0

18 juin 1993

PROJET SYNCHRONE
Les formats communs des langages synchrones

Résumé

Ce document définit la syntaxe et la sémantique des formats communs des langages synchrones. Ces formats constituent un socle commun à la programmation synchrone, sur lequel de nombreux outils seront applicables. Le socle est constitué de trois formats, entre lesquels existeront des passerelles de traduction :

- IC est un format parallèle de type *impératif;*
- GC est un format parallèle de type *flot de données;*
- OC est un format *automate* séquentiel.

SYNCHRON PROJECT
The common formats of synchronous languages

Abstract

This report defines syntax and semantics of the common formats of synchronous languages. The formats will support the largest part of the compiling process of these languages. There are three formats, between them translators will be defined:

- IC is a parallel format of *imperative* style;
- GC is a parallel format of *data-flow* style;
- OC is a sequential format to describe automata.

Avertissement : Ce document définit la première version, dite Version 0, des formats communs des langages synchrones. Des évolutions sont à prévoir pour les futures versions : elles résulteront notamment des expérimentations conduites autour de ces formats. D'autres évolutions sont d'ores et déjà envisagées : elles concernent par exemple la communication inter-formats, l'enrichissement des structures de données, etc.

Table des matières

Chapitre 1

Organisation générale

1.1 Introduction

De nombreux "objets" de la vie courante sont désormais contrôlés de façon informatique : usines, avions, TGV, voitures, télévisions... À partir de signaux reçus du monde extérieur, il faut élaborer des commandes en un temps généralement contraint. Traditionnellement, ce secteur des applications dites "temps-réel" utilisait des systèmes analogiques, des relais. Aujourd'hui, l'informatique et l'électronique intelligente sont embarquées dans tous ces produits. Or, le service essentiel demeure la sécurité : une seule erreur, logicielle ou matérielle, lors de la conception ou de la réalisation, peut avoir des conséquences dramatiques. La fiabilité du logiciel embarqué est donc critique : elle repose sur une solution exacte au problème à traiter et sur la rigueur dans le processus d'écriture des programmes. Ces deux aspects dépendent de manière cruciale du formalisme d'expression utilisé pour les spécifications, du langage de programmation supportant ce formalisme et des services annexes qui lui sont associés. Il faut donc offrir au développeur d'applications temps-réel un langage puissant, sous-tendu par un formalisme rigoureux, et muni d'environnements (compilateurs, vérificateurs, éditeurs graphiques) de qualité : c'est l'ambition des langages de programmation synchrones ESTEREL, LUSTRE, SIGNAL et ARGOS. Le caractère *synchrone*[1] de ces langages permet de garantir le bon déroulement des opérations du processus de conception/réalisation/maintenance.

Les domaines d'application des langages synchrones et de leurs outils et environnements associés couvrent toute la gamme des applications temps-réel, des micro-applications intégrées aux grands systèmes :

- *matériel :* circuiterie intégrée pour les applications temps-réel compactes (calcul rapide, automates, micro-contrôleurs) ; de tels besoins sont fréquents (automatismes, automobile, interfaces, traitement du signal...) ; les langages synchrones peuvent alors s'interfacer avec des chaînes de CAO de circuiterie ;

- *architectures multi-micro ou hétérogènes :* on les rencontre dans les applications militaires, commandes de procédés industriels, systèmes de traitement de l'information temps-réel... ; les langages synchrones permettent d'avoir une vision globale de l'application (spécification facilitée) et de produire du code pour ces architectures ;

[1] l'approche synchrone s'appuie sur une vision du temps qui est de nature "logique" et qui est globale à tout le système ; par opposition à l'approche asynchrone, cela permet des raisonnements formels sur les aspects synchronisation, logique et ordonnancement des calculs.

- *applications complexes* de type "systèmes", domaine où les SSII françaises sont très bien placées ainsi que les industriels réalisateurs de grands projets (THOMSON, ALSTHOM, MATRA, MERLIN-GERIN...) ; les langages synchrones fournissent le cœur d'une chaîne *spécification* → *implémentation* sans rupture.

S'adresser à une grande diversité de métiers à travers un concept unique nécessite le développement de dialogues et environnements adaptés à chaque culture particulière. La programmation synchrone doit donc cesser d'être une juxtaposition d'outils pour devenir un *concept,* permettant ainsi la multiplication des interventions sur un objet cohérent. Ceci justifie l'entreprise du projet SYNCHRONE : *définition et réalisation en exemplaire non unique d'un socle commun,* dont les spécifications générales font l'objet du présent document. Ce socle est le candidat idéal pour l'intégration dans de véritables ateliers de développement d'applications temps-réel dont il constituera une pièce essentielle. Il permettra de faire évoluer les langages *synchrones* de l'état de produits juxtaposés, voire concurrents, vers l'état de langages s'appuyant sur un format d'échange commun : les langages synchrones s'en trouveront mutuellement renforcés. Le résultat attendu du projet est montré à la figure 1.1 de la section 1.2.

1.2 Formats SYNCHRONE

Les quatre langages ESTEREL, LUSTRE, SIGNAL et ARGOS utilisent des styles différents : ESTEREL et ARGOS sont impératifs, LUSTRE est déclaratif de type fonctionnel, SIGNAL est un langage déclaratif de type relationnel. Ceci est une richesse, tant du point de vue de l'exploration théorique des possibilités de l'approche synchrone, que du point de vue de la diversité de culture des utilisateurs potentiels. En retour, cette diversité est en soi un handicap pour la dissémination du concept. Le socle commun de la programmation synchrone devrait permettre de combler ce dernier handicap, tout en préservant la richesse de la diversité.

Le socle commun de la programmation synchrone (cf. Fig. 1.1) consiste en un ensemble de formats, à partir desquels divers outils sont applicables. Le nécessaire compromis entre la facilité de traduction des langages source dans ces formats, et l'adéquation des formats aux traitements effectués par les outils en aval, a conduit à distinguer deux niveaux :

- un niveau où le parallélisme présent dans le programme source est préservé,
- un niveau de description purement séquentielle ;

et dans le premier niveau, deux styles de description : impératif et déclaratif.

Le socle est ainsi constitué de trois formats, entre lesquels existeront des "passerelles" de traduction :

IC : c'est un format intermédiaire synchrone de type *impératif :* il s'agit d'un format parallèle, permettant de lancer, d'enchaîner, d'interrompre, et de conduire concurremment des actions. Ce format est dérivé du code intermédiaire du compilateur ESTEREL actuel, qui a aussi été utilisée comme cible du compilateur ARGOS. Outre ESTEREL et ARGOS, ce format pourrait servir de format cible à divers formalismes de type impératif.

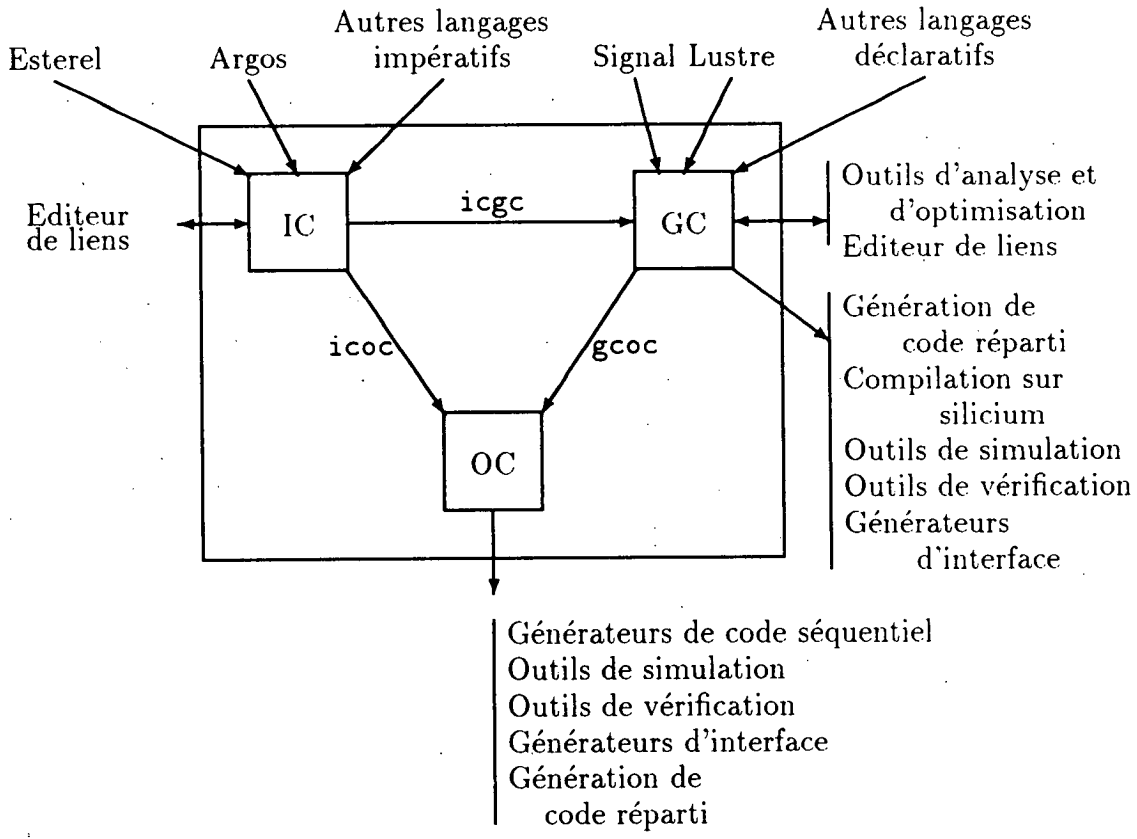


Figure 1.1 : Le socle commun de la programmation synchrone

GC : c'est un format intermédiaire synchrone de type *graphe* : il s'agit d'un format parallèle, décrivant dans un symbolisme graphique de type schéma-bloc des liens entre opérations, ces liens étant étiquetés par "l'horloge" à laquelle ils sont effectifs. Il s'agit d'un format hiérarchique (une boîte est soit une fonction prédéfinie, soit un schéma-bloc). Outre SIGNAL et LUSTRE, ce format pourrait servir de format cible à divers formalismes de type déclaratif. Il est en outre un candidat naturel pour la production de composants matériels, pour les mises en œuvre réparties, et pour les services d'évaluation de performance et de placement-routage sur une architecture distribuée. A ce titre, il doit aussi pouvoir faire partie des chaînes d'outils issues des langages impératifs : c'est pourquoi un programme GC peut contenir des structures impératives, permettant de décrire des actions connectées aux objets du schéma-bloc au moyen de *dépendances*, qui sont des directives de séquencement.

OC : c'est un format "automate" séquentiel, résultat des travaux communs sur les langages ESTEREL et LUSTRE. C'est un outil intéressant pour la production de code séquentiel (C, ADA, FORTRAN, langages à automate...). C'est l'outil privilégié pour les traitements et services propres aux automates (simulations, animations, preuves).

1.3 Organisation du document

Les formats communs sont des formats internes destinés à être produits et exploités par des outils automatiques. À la différence des langages, leur syntaxe est donc conçue surtout en vue d'une analyse automatique.

Quoique destinés à décrire des objets de natures assez différentes, les trois formats ont été définis de manière à partager au mieux les notations et les notions de base. Une partie essentiellement commune aux trois formats permet de décrire, sous forme de tables, les objets de base des systèmes spécifiés, ainsi que les objets importés. Ces objets importés sont décrits dans d'autres langages appelés *langages hôtes* : un langage hôte est soit un langage cible des compilateurs des langages synchrones, soit tout autre langage susceptible d'être interfacé avec le code produit.

Le chapitre 2 décrit les conventions générales et les tables communes aux trois formats. Le chapitre 3 décrit la partie commune aux deux formats intermédiaires impératifs des langages synchrones. Les chapitres suivants (4, 5, 6) décrivent ensuite les éléments spécifiques à chacun des formats.

Forme de présentation de la syntaxe

On distingue trois classes de termes pour la description de la syntaxe des formats :

- le vocabulaire du niveau lexical: les **terminaux** désignent chacun un ensemble énuméré de suites indivisibles de caractères;
- les structures lexicales: les **Terminaux** du niveau syntaxique, sont définis, à un niveau lexical, par des règles dans une grammaire dont le vocabulaire est l'union des ensembles **terminaux**; aucun caractère implicite (par exemple, séparateurs) n'est autorisé dans les termes construits selon ces règles;
- les stuctures syntaxiques: les **NON-TERMINAUX** sont définis, à un niveau syntaxique, par des règles dans une grammaire dont le vocabulaire est formé des ensembles **Terminaux**; entre deux **Terminaux**, on peut insérer un nombre quelconque de séparateurs.

Chaque unité des formats est introduite puis décrite à l'aide d'une grammaire. La grammaire donne la syntaxe hors-contexte de la structure considérée selon l'une des formes suivantes :

♠ **STRUCTURE** ::=

- DÉRIVATION1 •
- DÉRIVATION2 •
- ... •

♠ **Terminal** ::=

- DÉRIVATION1 •
- DÉRIVATION2 •
- ... •

♠ **terminal** ::=

• ENSEMBLE1 •
• ENSEMBLE2 •
• ... •

DÉRIVATION1, DÉRIVATION2 sont des réécritures de la variable **STRUCTURE** (respectivement, de la variable **Terminal**). ENSEMBLE1, ENSEMBLE2 sont des réécritures de la variable **terminal** ; ce sont des **Dérivations** réduites à un seul élément (cf ci-dessous).

Chaque DÉRIVATION est une suite d'*éléments* dont chacun peut être :

- un **ensemble** de caractères, noté dans cette typographie (niveau lexical uniquement),
- un **mot-clé** (formé de lettres), ou un **symbole** terminal (formé d'autres caractères admissibles), dans cette typographie,
- un **Terminal**, dans cette typographie,
- une **STRUCTURE** syntaxique, dans cette typographie (niveau syntaxique uniquement),
- une suite d'*éléments* avec ou sans séparateur, respectivement sous les formes suivantes :
  - { *élément* **symbole** ... }
  - { *élément* ... }
- un *élément* optionnel, noté [*élément*].
- une différence ensembliste, notée { *élément1* ⊖ *élément2* }, permettant de dériver les textes de *élément1* qui ne sont pas des textes de *élément2*.

Les structures syntaxiques ou lexicales pourront apparaître soit au pluriel, soit au singulier selon le contexte. Elles pourront être complétées par une *information contextuelle*, dans cette typographie. Enfin plusieurs dérivations pourront être placées sur une même ligne ; dans ce cas la marque de fin de chacune d'elles sera confondue avec la marque de début de sa suivante.

Chapitre 2

Les éléments communs

Ce chapitre décrit la partie commune des trois formats intermédiaires des langages synchrones. Ainsi, les quatre sections suivantes présentent respectivement :

- les unités lexicales,
- l'organisation générale des formats, en paquets, entités et tables,
- les pragmas,
- enfin, l'organisation commune des données dans les formats.

2.1 Unités lexicales

On présente successivement les caractères utilisés pour former les unités lexicales, puis les commentaires, les nombres, les chaînes de caractères, les identificateurs, et les mots-clés et identificateurs prédéfinis.

Les index, qui sont également reconnus comme des unités lexicales, sont présentés dans cette section.

2.1.1 Conventions lexicales

Un texte exprimé dans les formats communs des langages synchrones est une suite d'éléments du vocabulaire **Terminal** (cf. 1.3, page 4) des formats, entre lesquels peut apparaître un nombre quelconque (éventuellement nul) de **séparateurs**. Un **Terminal** des formats communs est la plus longue suite de **terminaux** contigus et un **terminal** la plus longue suite de caractères contigus qui peuvent être formées par une analyse de gauche à droite respectant les règles décrites dans cette section. Un terminal peut comporter une marque discriminante ; les marques suivantes ne sont pas des **caractères** :

♠ **marque-préfixe** ::= • `\` *début de* **CodeCaractère** •
• `$I` *début des identificateurs d'***objet***s prédéfinis impératifs* •
• `$g` *début des identificateurs d'***objet***s prédéfinis* GC •
• `--` *début de commentaire ligne* •

Les caractères utilisés dans les formats sont décrits dans la présente section (**Caractère**). Ils peuvent être désignés sous une forme codée qui n'est utilisable que dans les commentaires, les constantes chaînes et les pragmas.

♠ **Caractère** ::= • **caractère** • **CodeCaractère** •

Les caractères

L'ensemble des caractères (noté **caractère**) utilisés dans les formats contient les sous-ensembles suivants :

♠ **caractère** ::= • **car-ident** • **marque** • **délimiteur** • **séparateur** • **autre-caractère** •

1. L'ensemble **car-ident** des caractères utilisés pour construire les identificateurs :

   ♠ **car-ident** ::= • **car-lettre** • **car-chiffre** • [_] •

   ♠ **car-lettre** ::= • **car-lettre-majuscule** • **car-lettre-minuscule** •

   ♠ **car-lettre-majuscule** ::=

   • [A] • [B] • [C] • [D] • [E] • [F] • [G] • [H] • [I] •
   • [J] • [K] • [L] • [M] • [N] • [O] • [P] • [Q] • [R] •
   • [S] • [T] • [U] • [V] • [W] • [X] • [Y] • [Z] •

   ♠ **car-lettre-minuscule** ::=

   • [a] • [b] • [c] • [d] • [e] • [f] • [g] • [h] • [i] •
   • [j] • [k] • [l] • [m] • [n] • [o] • [p] • [q] • [r] •
   • [s] • [t] • [u] • [v] • [w] • [x] • [y] • [z] •

   ♠ **car-chiffre** ::= • [0] • [1] • [2] • [3] • [4] • [5] • [6] • [7] • [8] • [9] •

   Les formes majuscule et minuscule d'une même lettre (**car-lettre**) sont distinguées.

2. L'ensemble **marque** des caractères discriminants des unités lexicales :

♠ **marque** ::=
- • [$] *début des identificateurs d'***objet***s prédéfinis communs* •
- • [:] *terminaison des mots-clés* •
- • [.] *caractère séparateur dans les constantes réelles et les index* •
- • [+] *signe positif des exposants de constantes réelles* •
- • [−] *signe négatif des exposants de constantes réelles* •
- • [#] *début d'une constante littérale* •
- • [@] *début d'un index de constante* •
- • [!] *début de test de présence d'un signal* •
- • [?] *début d'accès à la valeur d'un signal* •
- • [”] *début et terminaison des chaînes* •
- • [%] *début et terminaison des pragmas* •

3. Les délimiteurs sont des terminaux du niveau syntaxique construits avec des caractères autres que lettres et chiffres :

♠ **délimiteur** ::= • **icgcocsymb** • **icocsymb** •

♠ **icgcocsymb** ::=
- • [(] • [)] • *imbrication*
- • [,] • *séparation de champs*
- • [/] • *substitution*
- • [:] • *marque de résultat de fonction et suffixe de numéro*
- • [;] • *fin de structure*

♠ **icocsymb** ::=
- • [<] • [>] • *groupement d'arguments*
- • [[] • []] • *groupement d'arguments*
- • [{] • [}] • *groupement d'arguments*

4. Les **séparateurs** donnés ici dans leur code hexadécimal ASCII :

♠ **séparateur** ::=
- • [\x9] *tabulation horizontale* •
- • [\xA] *nouvelle ligne* •
- • [\xC] *nouvelle page* •
- • [\xD] *retour chariot* •
- • [\x20] *espace* •

5. Les autres caractères *imprimables,* utilisables dans les commentaires, les pragmas et les dénotations de constantes. Ce sous-ensemble, **autre-caractère**, n'est pas défini par le format.

Les caractères codés

Tous les caractères (imprimables ou non imprimables) peuvent être désignés sous une forme codée (**CodeCaractère**) dans les commentaires, les constantes chaînes et les prag-

mas. Les codes autorisés sont ceux de la norme ANSI du langage C à laquelle a été ajouté le caractère d'échappement [\%] utilisé dans les pragmas. Un caractère sous forme codée est soit un caractère spécial (**code-échappement**), soit un caractère codé sous forme octale (**CodeOctal**), soit un caractère codé sous forme hexadécimale (**CodeHexadécimal**).

♠ **CodeCaractère** ::= • **CodeOctal** • **CodeHexadécimal** •
• **code-échappement** •

♠ **CodeOctal** ::= • [\ ] **car-octal** [**car-octal** [**car-octal**]] •

♠ **car-octal** ::= • [0] • [1] • [2] • [3] • [4] • [5] • [6] • [7] •

♠ **CodeHexadécimal** ::= • [\x] **car-hexadécimal** [**car-hexadécimal**] •

♠ **car-hexadécimal** ::= • **car-chiffre** •
• [A] • [B] • [C] • [D] • [E] • [F] •
• [a] • [b] • [c] • [d] • [e] • [f] •

♠ **code-échappement** ::= • [\a] *signal sonore* •
• [\b] *espace arrière* •
• [\f] *saut de page* •
• [\n] *saut de ligne* •
• [\r] *retour chariot* •
• [\t] *tabulation horizontale* •
• [\v] *tabulation verticale* •
• [\\ ] *anti-slash* •
• [\"] *double quote* •
• [\'] *simple quote* •
• [\?] *point d'interrogation* •
• [\%] *pourcent* •

2.1.2 Commentaires

Tout fragment de texte compris entre le symbole [--] et la fin de ligne est un commentaire mono-ligne.

D'autre part, on peut également mettre des commentaires sous forme de pragma (cf. 2.3, page 20).

2.1.3 Nombres

Deux types de nombres sont gérés dans les formats communs : les entiers naturels positifs et les flottants positifs (codés en simple ou double précision). Les nombres négatifs sont obtenus par application de la fonction **$uminus** du type considéré (cf. 2.4.3, page 24).

Entiers

Un entier est une unité lexicale exclusivement composée d'une suite de **car-chiffres** interprétée en base 10.

Une valeur entière est désignée par une **Cst-entière**.

♠ **Cst-entière** ::= • { **car-chiffre** ... } •

Réels

Un flottant en simple précision est une unité lexicale composée de la succession :

- d'un entier ;
- d'une partie décimale, constituée d'un point (.) suivi par un entier ;
- d'un exposant, constitué du caractère **E** ou **e**, suivi d'un signe (caractère + optionnel ou −) et d'un entier.

Un flottant en double précision est une unité lexicale dont la description est identique à celle d'un flottant en simple précision à l'exception de l'exposant. En effet, celui-ci diffère par l'utilisation du caractère **D** ou **d** à la place du caractère **E** ou **e**.

Une **Cst-réelle** dénote la valeur approchée d'un nombre réel.

♠ **Cst-réelle** ::= • **Cst-réelle-simple-précision** •
• **Cst-réelle-double-précision** •

♠ **Cst-réelle-simple-précision** ::= • **Partie-fraction Exposant-simple-précision** •

♠ **Cst-réelle-double-précision** ::= • **Partie-fraction Exposant-double-précision** •

♠ **Partie-fraction** ::= • **Cst-entière** . **Cst-entière** •

♠ **Exposant-simple-précision** ::= • **e** **Cst-relative** • **E** **Cst-relative** •

♠ **Exposant-double-précision** ::= • **d** **Cst-relative** • **D** **Cst-relative** •

♠ **Cst-relative** ::= • **Cst-entière** • + **Cst-entière** •
• − **Cst-entière** •

2.1.4 Chaînes de caractères

Les chaînes de caractères sont les chaînes de caractères de la norme ANSI du langage C.

Une valeur de chaîne est dénotée par une **Cst-chaîne**.

♠ **Cst-chaîne** ::= • " [{ **CaractèreChaîne** ... }] " •

♠ **CaractèreChaîne** ::= • { **Caractère** ⊖ **car-spec-chaîne** } •

♠ **car-spec-chaîne** ::= • ["] • [\ ] •

2.1.5 Identificateurs

Les identificateurs servent à désigner les objets. Parmi les identificateurs, on distingue les identificateurs d'**objet**s prédéfinis.

Les identificateurs sont des unités lexicales composées de caractères pris parmi l'ensemble formé des **car-lettres**, du caractère [_] et des **car-chiffres** ; un identificateur ne peut commencer par un **car-chiffre**. Ces identificateurs correspondent, par exemple, aux noms fournis par le programmeur dans un code source.

Les identificateurs d'**objet**s prédéfinis sont des unités lexicales commençant par le caractère [$], suivi d'un identificateur tel qu'indiqué ci-dessus.

♠ **Identificateur** ::= • **car-début-ident** [{ **car-ident** ... }] •
• [$] **car-début-ident** [{ **car-ident** ... }] •

♠ **car-début-ident** ::= • { **car-ident** ⊖ **car-chiffre** } •

On pourra dans ce document désigner un **Identificateur** d'une catégorie d'**objet**s particulière X par **Identificateur**-X.

2.1.6 Mots-clés et identificateurs prédéfinis

Mots-clés

Les mots-clés sont des unités lexicales commençant par une **car-lettre**, suivie de **car-lettres**, de **car-chiffres** ou du caractère [_] et se terminant par le caractère [:].

Les **Mots-clés** utilisés par les formats sont les suivants :

Communs à IC, GC et OC

♠ **icgcockw** ::=

• [call:] • [computed:] • [constants:] • [data:] • [dc:] •
• [end:] • [enddata:] • [endpackage:] • [flat:] • [functions:] •
• [gc:] • [i:] • [ic:] • [io:] • [import:] • [memorysafe:] •
• [o:] • [oc:] • [package:] • [pragmas:] • [procedures:] •
• [safe:] • [types:] • [unsafe:] • [value:] •

Communs à IC et OC

♠ **icockw** ::=

• act: • actions: • bool: • combine: •
• endmodule: • execs: • if: • input: • inputoutput: • instances: •
• kill: • local: • module: • multiple: •
• output: • relations: • reset: • resume: • return: • root: •
• signals: • single: • start: • startpoint: • suspend: •
• tasks: • variables: •

Spécifiques à GC

♠ **gckw** ::=

• assertions: • define: • definitions: • dependences: • do: •
• endinterface: • endnode: • extern: • flows: •
• interface: • node: • set: • synchronizations: •

Spécifiques à IC

♠ **ickw** ::=

• Access: • Action: • Emit: • Exit: • Fork: •
• Goloop: • Goto: • Halt: • Parallel: •
• Reset: • Return: • Run: • Stay: • Test: • Watchdog: •
• exitlevels: • goloops: • instructions: • linked: •

Spécifiques à OC

♠ **ockw** ::=

• calls: • dags: • sink: • states: •

Désignations de pragmas

♠ **pragmaskw** ::=

• alias: • awaited: • comment: • count: • dir: •
• emitted: • file: • haltset: • instance: • killed: •
• lc: • main: • name: • previous: • register: •
• sigbool: • sigval: • started: • wire: •

Identificateurs prédéfinis

Les **identificateurs-prédéfinis** dans les formats sont les suivants :

Communs à IC, GC et OC

♠ **icgcocui** ::=

• $and • $boolean • $cond • $conv • $div • $double •
• $eq • $false • $float • $ge • $gt • $integer •
• $le • $lt • $minus • $mod •
• $ne • $not • $or • $plus • $pure •
• $string • $times • $top • $true • $uminus •

Communs à IC et OC

♠ **icocui** ::=

• $assign • $dsz • $left_and • $left_or •

Spécifiques à GC

♠ **gcui** ::=

• $any • $base •
• $clkadd • $clkdiff • $clkeq • $clkimplies • $clkmult • $clkmutex •
• $clock • $default • $fby • $pre • $present •
• $select • $tt • $when • $win • $window •

2.2 Structuration commune des formats

Les formats communs permettent la description d'*entités*. Ces entités sont organisées en *paquets*, sans que cette organisation ait une fonction autre que documentaire dans un atelier d'accueil.

Un paquet est ainsi un univers synchrone, éventuellement hybride IC/GC/OC, constitué d'un ensemble d'*entités* numérotées. Les entités contenues dans un paquet peuvent être des blocs de données, des modules IC, des modules OC, des nœuds GC ou des interfaces GC.

Une entité est formée de *tables* qui décrivent les **objets** des formats.

2.2.1 Paquets

Un paquet comporte successivement les structures suivantes :

1. le mot-clé **package:** ;
2. le nombre d'entités décrites dans le paquet ;
3. l'identificateur du paquet ;
4. la liste des entités. Chaque entité est précédée d'un numéro (suivi du caractère :) qui permet de la désigner dans le paquet. Les entités composant le paquet

sont listées les unes à la suite des autres, dans l'ordre croissant de leurs numéros. Cette numérotation, sans trous, commence à 0. L'ordre des entités d'un paquet est tel qu'en l'absence de références croisées, il n'existe pas de références en avant (i.e., à un objet contenu dans une entité de numéro supérieur) pour les objets des formats autres que les pragmas ;

5. le mot-clé **endpackage:** qui termine le paquet.

Il ne peut y avoir deux paquets distincts de même nom dans un texte du format commun.

♠ **PAQUET** ::= • **package:** **EN-TÊTE-PAQUET**
{ **ENTRÉE-PAQUET** ... } **endpackage:** •

♠ **EN-TÊTE-PAQUET** ::= • **Nombre-entités** **Identificateur-***paquet* •

♠ **Nombre-entités** ::= • **Cst-entière** •

♠ **ENTRÉE-PAQUET** ::= • **Numéro-entité** **:** **ENTITÉ** •

♠ **Numéro-entité** ::= • **Cst-entière** •

2.2.2 Entités

L'entité est le constituant d'un paquet. Une entité est formée d'une suite de tables dans un ordre déterminé par la grammaire de chaque entité *(on trouve au plus une table par catégorie d'objets dans chaque entité).*

Les informations communes suivantes sont associées à chaque entité :

- son format suivi d'un numéro de version puis du nom de l'entité. Les formats actuellement autorisés (et décrits dans ce document) sont :
  - **dc:**, pour les blocs de données (cf. 2.4.6, page 27) ;
  - **ic:**, pour les modules IC (cf. 4, page 39) ;
  - **gc:**, pour les nœuds et interfaces GC (cf. 5, page 59) ;
  - **oc:**, pour les modules OC (cf. 6, page 83) ;
- un attribut optionnel (qui ne concerne pas les nœuds externes) dénoté par le mot-clé **flat:** ; cet attribut caractérise la forme des expressions (cf. 2.2.5, page 19), ainsi que celle des *dags* (cf. 6.5, page 87), présents dans l'entité ; l'attribut **flat:** indique que les expressions et les *dags* de l'entité sont plats, c'est-à-dire non imbriqués. Par défaut, l'attribut **flat:** est absent et expressions et *dags* sont admis dans toute leur généralité ;
- un attribut optionnel (qui ne concerne pas les blocs de données) précisant des propriétés comportementales de l'entité (cf. 2.4.5, page 26).

Dans la suite, sont décrites les entités communes aux formats (les blocs de données) et les entités spécifiques (les modules IC, les interfaces GC, les nœuds GC, les modules OC). La syntaxe de chacune de ces entités est donnée dans la description de l'entité.

♠ **ENTITÉ** ::= • **BLOC-DE-DONNÉES** •
• **MODULE-IC** •
• **INTERFACE** •
• **NŒUD** •
• **MODULE-OC** •

Il ne peut y avoir deux objets distincts de même nom dans une entité.

2.2.3 Tables

La table est le moyen universel pour décrire les **objets** dans le format. Toutes les tables ont la même structure. Une table comporte successivement :

1. un mot-clé identifiant la catégorie des **objets** qui la composent (par exemple **functions:**, **signals:**, etc.) ;
2. le nombre d'**objets** qu'elle contient (celui-ci peut être nul si la table ne comporte aucune entrée) ;
3. des informations spécifiques à chaque catégorie de table ;
4. la liste des **objets**. Chaque objet est précédé d'un numéro (suivi du caractère **:**) qui permet de le désigner ; il est suivi par le caractère **;**. Les objets composant la table sont listés les uns à la suite des autres, dans l'ordre croissant de leurs numéros. Cette numérotation *sans trous* commence à 0 ;
5. le mot-clé **end:** qui termine la table.

Afin d'éviter de la redécrire systématiquement, nous donnons ci-dessous une syntaxe générique pour les tables. Ainsi, le suffixe **X** désigne de manière générique les différentes catégories possibles de tables (il pourra apparaître au singulier ou au pluriel, en majuscules ou minuscules).

♠ **TABLE-X** ::= • **Nom-table-X** **EN-TÊTE-TABLE-X**
[{ **ENTRÉE-TABLE-X** ... }] **end:** •

♠ **EN-TÊTE-TABLE-X** ::= • **Nombre-entrées** [**INFOS-TABLE-X**] •

♠ **Nombre-entrées** ::= • **Cst-entière** •

♠ **ENTRÉE-TABLE-X** ::= • **Numéro-entrée** **:** **OBJET-X** **;** •

♠ **Numéro-entrée** ::= • **Cst-entière** •

Dans les tables d'**objet**s prédéfinis, chaque numéro d'entrée est précédé du préfixe servant à la désignation des **objet**s prédéfinis de la table (cf. 2.2.4, page 17) :

♠ **ENTRÉE-TABLE-X** ::= • **Index-objet-prédéfini** [:] **OBJET-X** [;] •

Les tables dont les noms sont indiqués ci-dessous sont les tables **X** communes aux formats :

♠ **Nom-table-importations** ::= • [**import:**] •

♠ **Nom-table-types** ::= • [**types:**] •

♠ **Nom-table-constantes** ::= • [**constants:**] •

♠ **Nom-table-fonctions** ::= • [**functions:**] •

♠ **Nom-table-procédures** ::= • [**procedures:**] •

♠ **Nom-table-pragmas** ::= • [**pragmas:**] •

Pour chaque catégorie **X** d'**objet**s, sont décrits dans la suite :

- les **INFOS-TABLE-X** s'il y a lieu (lorsqu'elles ne sont pas décrites pour une catégorie de tables **X**, les **INFOS-TABLE-X** sont vides ; lorsqu'elles sont présentes, chacune des informations supplémentaires doit se terminer par le caractère [;]) ;
- les **OBJETS-X**.

2.2.4 Index et mécanismes d'adressage

Index d'objet L'adressage d'un **objet** dans un paquet se fait par un index. Un index peut être un index d'**objet** prédéfini, un index local ou un index composé.

- Un *index d'***objet** *prédéfini* est composé :
  - soit du caractère [**$**] suivi du numéro de l'**objet** désigné appartenant à une table d'**objet**s prédéfinis communs aux trois formats ;
  - soit des caractères [**$I**] suivis d'un index spécifique pour les objets prédéfinis communs aux formats impératifs IC et OC ;
  - soit des caractères [**$g**] suivis d'un index spécifique pour les objets prédéfinis propres au format GC.
- Un *index local* est formé du numéro de l'**objet** désigné dans une table. Il permet de référencer, à l'intérieur d'une entité, un **objet** décrit dans l'une des tables de cette entité.
- Un *index composé* est formé :

- d'un index dans la table d'importation de l'entité courante, dans laquelle on trouve le numéro de l'entité — index externe —, contenant l'**objet** désigné,
- suivi du caractère [.], suivi d'un index local dans une table de l'entité désignée par l'index externe.

Dans un paquet, il permet de référencer par "adressage virtuel", à l'intérieur d'une entité, un **objet** décrit dans une table d'une autre entité.

- Un *index indéfini* est dénoté par le caractère [-]. Il permet d'omettre un index dans une position syntaxique où un index est attendu.

La catégorie d'un **objet** référencé au moyen d'un index est déterminée par le contexte d'utilisation de cet index.

Index externe

- Un index externe est :
  - soit un numéro d'entité dans le paquet courant,
  - soit un identificateur du paquet contenant l'entité désignée, suivi du caractère [.] puis du numéro de cette entité dans ce paquet.

Syntaxe des index

♠ **Index** ::= • **Index-local** •
• **Index-objet-prédéfini** •
• **Index-composé** •
• **Index-indéfini** •

♠ **Index-local** ::= • **Numéro-entrée** •

♠ **Index-objet-prédéfini** ::= • [$] **Cst-entière** •
• [$I] **Cst-entière** •
• [$g] **Cst-entière** •

♠ **Index-composé** ::= • **Index-importation** [.] **Index-local** •

♠ **Index-importation** ::= • **Cst-entière** •

♠ **Index-indéfini** ::= • [-] •

On pourra dans ce document désigner un index d'une catégorie d'**objet**s particulière X par **Index-**X.

Table des importations La table des importations présente dans une entité E offre un mécanisme d'accès associatif aux autres entités. On y trouve la liste des index externes des entités utilisées dans l'entité E ; ces utilisations se font sous la forme d'un indice dans la table des importations.

On rappelle que la syntaxe générique des tables est décrite en 2.2.3.

♠ **Objet-importation** ::= • **Numéro-entité** •
• **Identificateur-***paquet* [.] **Numéro-entité** •

2.2.5 Syntaxe commune des expressions

Une expression est une combinaison d'atomes de type prédéfini (cf. 2.4.1, page 22), de références à des constantes (cf. 2.4.2, page 24), d'appels de fonctions (cf. 2.4.3, page 24), et, selon le contexte d'utilisation, d'expressions spécifiques aux formats impératifs (cf. 3.1.3, page 31) ou d'expressions spécifiques GC (cf. 5.2.2, page 61).

Les **Atomes** sont constitués de deux unités lexicales : le caractère [#] immédiatement suivi d'une chaîne de caractères ou d'un nombre. Par exemple `#12` représente l'entier 12 et `#"Hello"` représente la chaîne `"Hello"`.

Une référence à une constante (**RefConstante**) est formée du caractère [@] immédiatement suivi de l'index de la constante.

Les appels de fonctions comportent l'index de la fonction, suivi, entre parenthèses, d'une liste d'expressions (séparées par des virgules) correspondant aux paramètres de la fonction.

♠ **EXPRESSION** ::= • **Terme** •
• **APPEL-FONCTION** •

♠ **Terme** ::= • **Atome** •
• **RefConstante** •
• **Terme-IC-OC** •
• **Terme-GC** •

♠ **Atome** ::= • [#] **Cst-chaîne** •
• [#] **Cst-entière** •
• [#] **Cst-réelle** •

♠ **RefConstante** ::= • [@] **Index-***constante* •

♠ **APPEL-FONCTION** ::=
• **Index-***fonction* [( ] [{ **EXPRESSION** [,] ... }] [ )] •

Les expressions ne peuvent être imbriquées (**EXPRESSION-PLATE**) dans un paquet auquel est associé l'attribut [**flat:**].

♠ **EXPRESSION-PLATE** ::= • **Terme** •
• **APPEL-FONCTION-PLATE** •

♠ **APPEL-FONCTION-PLATE** ::= • **Index-***fonction* [( ] [{ **Terme** [,] ... }] [ )] •

Pour comprendre les expressions données ci-dessous en exemple, on se reportera aux fonctions prédéfinies listées page 25.

Pour un objet `X` de type entier et d'index 3, qui est soit une variable (dans un module), soit un flot (dans un nœud) :

- `$14(3,#1)` a pour valeur le résultat de l'addition entière de la constante entière 1 à `X` ;

- `$10(@2,3)` a pour valeur *true* si la constante entière d'index 2 est strictement plus petite que `X`, *false* sinon.

2.3 Pragmas

Les pragmas sont des "commentaires exécutables", spécifiques à certains langages ou à certains outils.

2.3.1 Syntaxe des pragmas

Toute partie de texte entre deux **marques** [%] est un pragma. Un pragma est composé soit d'une suite de **Caractères** et de chaînes de caractères, soit d'un index dans une table de pragmas (index local pour l'entité courante, composé pour une entité importée). Pour faire figurer le caractère [%] dans un pragma (cela ne concerne donc pas les caractères [%] situés dans des chaînes de caractères à l'intérieur d'un pragma), on écrit [\%]. Pour faire figurer le caractère ["] dans un pragma, on écrit [\"]. Les pragmas ne peuvent pas être imbriqués.

Il existe deux types de pragmas :

- le pragma "référence dans la table des pragmas" est constitué d'un index entre [%] ;

- les autres pragmas sont reconnus par un mot-clé suivant immédiatement le premier [%] : sa lecture permet à chaque outil de savoir s'il est concerné par le pragma, ou s'il doit le traiter comme un simple commentaire. La "valeur" du pragma, composée d'unités lexicales différentes d'un pragma, est donnée par le texte qui suit le mot-clé jusqu'au [%] de clôture.

♠ **Pragma** ::= • **Pragma-référence** • **Pragma-standard** •

♠ **Pragma-référence** ::= • [%] **Index-*pragmas*** [%] •

♠ **Pragma-standard** ::= • [%] **Nom-pragma Valeur-pragma** [%] •

♠ **Valeur-pragma** ::= • [{ **Unité-pragma** ... }] •

♠ **Unité-pragma** ::= • **CaractèrePragma** • **Cst-chaîne** •

♠ **CaractèrePragma** ::= • { **Caractère** ⊖ **car-spec-pragma** } •

♠ **car-spec-pragma** ::= • [%] • **car-spec-chaîne** •

2.3.2 Table des pragmas et pragmas calculés

Une table de pragmas commence par le mot-clé **pragmas:** suivi par le nombre d'entrées dans la table. Ensuite, l'en-tête contient le mot-clé **computed:** suivi d'une liste (éventuellement vide) de mots-clés identificateurs de pragmas et du caractère ; . Cette liste indique les pragmas que l'outil qui a généré le code a su calculer. Une structure syntaxique à laquelle peut être associé (d'après la définition du pragma) un pragma de cette liste en est donc automatiquement muni, soit avec une valeur explicite (pragma dans le code), soit avec la valeur par défaut spécifiée lors de la définition du pragma (par exemple, une liste vide pour les pragmas dont la valeur est une liste).

La table contient ensuite la liste des entrées et se termine par **end:**. Chaque entrée dans la table est constituée de son index, suivi de : , d'une suite de pragmas (avec leurs délimiteurs %) et du caractère ; .

♠ **INFOS-TABLE-PRAGMAS** ::=

• **computed:** [{ **Nom-pragma** ... }] ; •

♠ **OBJET-PRAGMAS** ::= • { **Pragma** ... } •

Les outils qui traduisent un format en un autre ou qui manipulent le code d'un format donné (ex: compilateurs, optimiseurs) doivent, dans la mesure du possible, transmettre les pragmas non reconnus tels quels et supprimer les noms des éventuels pragmas non reconnus de la liste des pragmas calculés. En effet, si un outil ne connaît pas un pragma donné, il ne peut pas affirmer que ce pragma a été correctement calculé pour toutes les structures syntaxiques auxquelles il peut être associé et en particulier pour celles ajoutées ou modifiées par cet outil.

2.3.3 Règles d'association des pragmas

Les pragmas peuvent apparaître à n'importe quel endroit dans un paquet.

Les règles d'association des pragmas sont différentes selon que le pragma se trouve dans une table ou non.

- Un pragma qui ne se trouve pas dans une table est associé à l'entité (bloc de données, module, nœud ou interface) qui le contient directement ou, si aucune entité ne le contient, au paquet dans lequel il se trouve.
- Un pragma dans une table est associé à l'élément qui le précède immédiatement dans cette table selon les règles suivantes :
  - un pragma placé entre le mot-clé identifiant la table et le mot-clé identifiant la première information qu'elle contient ou par défaut la première entrée, ou par défaut la fin de la table (table ne contenant aucun **objet**), est associé à la table ;
  - un pragma placé immédiatement après le caractère ; terminant une information ou une entrée de la table est associé à cette information ou entrée ;
  - un pragma suivant immédiatement un élément (abstraction faite des séparateurs) d'une entrée ou d'une information de la table, est associé à cet élément.

2.3.4 Procédure de définition d'un pragma

La définition d'un pragma consiste en la donnée des six éléments suivants :

- le mot-clé identifiant le pragma (également appelé "nom du pragma") ;
- la liste des objets auxquels le pragma peut être associé ;
- la syntaxe précise de la valeur du pragma (**Valeur-pragma**) ;
- la signification du pragma, éventuellement fonction de l'objet auquel il est associé ;
- l'autorisation ou non de faire figurer le nom de ce pragma (c'est-à-dire son mot-clé) dans la liste des pragmas calculés d'une entité ;
- si le nom du pragma peut figurer dans la liste des pragmas calculés d'une entité, la valeur par défaut du pragma.

Afin d'éviter les collisions de nom et pour définir le contenu des pragmas, un "BUREAU D'ENREGISTREMENT DES PRAGMAS STANDARDS" aura la charge d'entériner les demandes. Ces demandes devront comporter les six (ou cinq s'il n'y a pas de valeur par défaut) éléments ci-dessus pour chaque pragma.

L'annexe A de ce document définit un certain nombre de pragmas, en particulier le pragma commentaire, et constitue les premières demandes d'enregistrement auprès du BUREAU D'ENREGISTREMENT DES PRAGMAS STANDARDS.

2.4 Organisation commune des données

La description des domaines de valeur (des éléments des suites synchrones) et des opérations qu'on leur applique est organisée en tables dans des blocs de données. Nous décrivons successivement les types communs, les constantes, les fonctions et les procédures sur ces types, et les blocs de données qui les contiennent.

Utilisées en particulier pour construire des interfaces avec des environnements d'accueil non nécessairement synchrones, les procédures et les fonctions citées dans les blocs de données doivent respecter des règles permettant d'en assurer un bon usage par un traitement synchrone. Afin de caractériser les effets possibles de leur invocation un attribut sémantique (cf. 2.4.5, page 26) est associé à leur déclaration. Cet attribut peut également caractériser des nœuds et des modules.

2.4.1 Types

Un descripteur de type se réduit à un identificateur en donnant le nom.

À chaque type peuvent être associées des procédures et fonctions permettant de traiter les valeurs de ce type. Parmi ces fonctions et procédures on distingue :

- une procédure d'affectation de ce type. Cette procédure prend un paramètre d'entrée et un paramètre de sortie du type donné ; elle affecte la valeur du paramètre d'entrée au paramètre de sortie. Cette procédure aura le nom prédéfini **`$assign`** ;

- une fonction d'égalité de ce type. Cette fonction prend deux arguments du type donné et rend le booléen vrai (constante `$2`) si et seulement si ses arguments sont égaux et faux (constante `$1`) sinon. Cette fonction aura le nom prédéfini `$eq` ; cette fonction doit être *safe* (cf. 2.4.5, page 26) ;
- une fonction différence de ce type. Cette fonction prend deux arguments du type donné et rend pour résultat la négation booléenne du résultat de `$eq`. Cette fonction aura le nom prédéfini `$ne` ; cette fonction doit être *safe* ;
- une fonction conditionnelle de ce type. Cette fonction prend un argument de type booléen (`$1`) et deux arguments du type donné ; elle rend le premier des deux objets si l'argument de type booléen est vrai (constante `$2`) et le second objet sinon. Cette fonction aura le nom prédéfini `$cond` ; cette fonction doit être *safe*.

Remarque : Si, pour un type donné, la procédure d'affectation ou une de ces fonctions n'est pas utilisée, elle n'est pas nécessairement présente dans la table correspondante.

♠ **OBJET-TYPE** ::= • **Identificateur-***type* •

Les types prédéfinis communs sont donnés à la table 2.1 ; la procédure d'affectation et les fonctions d'égalité, de différence et conditionnelle associées apparaissent en commentaire dans cette table. Les fonctions et procédures prédéfinies associées aux types prédéfinis communs sont détaillées aux tables 2.5 et 3.1.

```
types: 6
$0: $pure;    --                    $0  code $eq
$1: $boolean; -- $0 code $assign, $1  code $eq, $2  code $ne, $3  code $cond
$2: $integer; -- $1 code $assign, $7  code $eq, $8  code $ne, $9  code $cond
$3: $string;  -- $2 code $assign, $20 code $eq, $21 code $ne, $22 code $cond
$4: $float;   -- $3 code $assign, $23 code $eq, $24 code $ne, $25 code $cond
$5: $double;  -- $4 code $assign, $35 code $eq, $36 code $ne, $37 code $cond
end:
```

Tableau 2.1 : Types prédéfinis communs

```
types: 1
0: A_TYPE;  -- 0 code $assign, 0 code $eq, 1 code $ne, 2 code $cond
end:
```

Tableau 2.2 : Une table de type

Le type booléen (`$1`) possède toutes les propriétés algébriques standards des booléens. Les constantes booléennes `$true` (`$2`) et `$false` (`$1`) sont prédéfinies dans la table des constantes (cf. 2.4.2, page 24).

Le type `$pure` (`$0`) possède une valeur unique dénotée par la constante prédéfinie `$top` d'index `$0` (cf. 2.4.2, page 24).

Les fonctions d'égalité et de différence sont supposées être cohérentes (c'est-à-dire que pour deux objets quelconques, une et une seule de ces deux fonctions doit renvoyer la constante booléenne vraie $2). Le remplacement de la négation d'une égalité par une différence laisse invariante la sémantique du programme.

2.4.2 Constantes

Un descripteur de constante comporte le nom de la constante suivi d'un index donnant son type. Vient ensuite un champ optionnel repéré par le mot-clé **value:** définissant par une expression la valeur de la constante.

♠ **OBJET-CONSTANTE** ::=

• **Identificateur-***constante* **Index-***type* [**VALEUR-CONSTANTE**] •

♠ **VALEUR-CONSTANTE** ::= • **value:** **EXPRESSION** •

```
constants: 3
$0: $top $0;
$1: $false $1;
$2: $true $1;
end:
```

Tableau 2.3 : Les constantes prédéfinies communes

```
constants: 3
0: LINESIZE $2;
1: COLUMNSIZE $2;
2: ARRAYSIZE $2 value: $16(@0,@1);
end:
```

Tableau 2.4 : Une table de constantes

2.4.3 Fonctions

Une fonction est une structure syntaxique permettant, au sein d'expressions, l'invocation positionnelle de fonctions au sens usuel des langages hôtes. La description d'une fonction contient, dans l'ordre :

- un identificateur donnant le nom de la fonction ;
- la liste des index des types des arguments (entre parenthèses et séparés par des virgules) ;
- l'index du type de la valeur retournée précédé de **:** ;
- éventuellement, les mots-clés **memorysafe:** ou **unsafe:** (cf. 2.4.5, page 26), en l'absence desquels la fonction est supposée *safe*.

♠ **OBJET-FONCTION** ::=

• **Identificateur-***fonction* **(** [{ **Index-***type* **,** ... }] **)** **:** **Index-***type* [**Propriété-de-processus**] •

Rappelons que pour un type μ quelconque, peuvent exister les fonctions de noms prédéfinis **$eq**, **$ne** et **$cond**.
Exemple de table de fonctions :

```
functions: 5
0: $eq (0, 0): $1;
1: $ne (0, 0): $1;
2: $cond ($1, 0, 0): 0;
3: MODULO ($2, $2): $2;
4: RANDOM (): $4 unsafe:;
end:
```

Dans la description précédente, le nom de la fonction utilisateur (comme `MODULO`) est un identificateur. Les fonctions de manipulation des types dont les index apparaissent dans la table des types ont des noms prédéfinis. La fonction `RANDOM`, sans argument, retourne une valeur de type flottant prédéfini `$4` qui, d'après le mot-clé `unsafe:`, peut différer d'un appel à l'autre.

La table 2.5 donne la liste des fonctions prédéfinies communes. Une référence à une

```
functions: 50
-- pure
 $0: $eq ($0, $0): $1;
-- boolean
 $1: $eq ($1, $1): $1;
 $2: $ne ($1, $1): $1;
 $3: $cond ($1, $1, $1): $1;
 $4: $or ($1, $1): $1;
 $5: $and ($1, $1): $1;
 $6: $not ($1): $1;
-- integer
 $7: $eq ($2, $2): $1;
 $8: $ne ($2, $2): $1;
 $9: $cond ($1, $2, $2): $2;
$10: $lt ($2, $2): $1;
$11: $le ($2, $2): $1;
$12: $gt ($2, $2): $1;
$13: $ge ($2, $2): $1;
$14: $plus ($2, $2): $2;
$15: $minus ($2, $2): $2;
$16: $times ($2, $2): $2;
$17: $mod ($2, $2): $2;
$18: $div ($2, $2): $2;
$19: $uminus ($2): $2;
-- string
$20: $eq ($3, $3): $1;
$21: $ne ($3, $3): $1;
$22: $cond ($1, $3, $3): $3;
-- float
$23: $eq ($4, $4): $1;
$24: $ne ($4, $4): $1;
$25: $cond ($1, $4, $4): $4;
$26: $lt ($4, $4): $1;
$27: $le ($4, $4): $1;
$28: $gt ($4, $4): $1;
$29: $ge ($4, $4): $1;
$30: $plus ($4, $4): $4;
$31: $minus ($4, $4): $4;
$32: $times ($4, $4): $4;
$33: $div ($4, $4): $4;
$34: $uminus ($4): $4;
-- double
$35: $eq ($5, $5): $1;
$36: $ne ($5, $5): $1;
$37: $cond ($1, $5, $5): $5;
$38: $lt ($5, $5): $1;
$39: $le ($5, $5): $1;
$40: $gt ($5, $5): $1;
$41: $ge ($5, $5): $1;
$42: $plus ($5, $5): $5;
$43: $minus ($5, $5): $5;
$44: $times ($5, $5): $5;
$45: $div ($5, $5): $5;
$46: $uminus ($5): $5;
-- conversions
$47: $conv ($2): $4;
$48: $conv ($2): $5;
$49: $conv ($4): $5;
end:
```

Tableau 2.5 : Les fonctions prédéfinies communes

de ces fonctions est l'**Index**-*objet-prédéfini* formé du symbole `$`, suivi de son numéro dans cette table.

Ces fonctions sont strictes. Outre les fonctions `$eq`, `$ne` et `$cond`, on y trouve :

- les fonctions de comparaison de valeurs numériques (types `$integer`, `$float` et `$double`)
  - `$lt` dont le résultat est vrai si le premier argument est strictement plus petit que le second, faux sinon,
  - `$le` dont le résultat est vrai si le premier argument est au plus égal au second, faux sinon,
  - `$gt` dont le résultat est vrai si le premier argument est strictement plus grand que le second, faux sinon,
  - `$ge` dont le résultat est vrai si le premier argument est au moins égal au second, faux sinon ;

- les fonctions d'addition (`$plus`), de soustraction (`$minus`), de multiplication (`$times`), de division (`$div`) et d'inversion de signe (`$uminus`) pour les valeurs numériques ;
- la fonction modulo (`$mod`) pour les valeurs entières ;
- les fonctions de conversion (`$conv`) successivement de `$integer` à `$float`, de `$integer` à `$double` et de `$float` à `$double`.

2.4.4 Procédures

Une procédure est une structure syntaxique permettant l'invocation positionnelle des procédures au sens usuel des langages hôtes. Une procédure est invoquée par un appel de procédure (`call:`).

La description d'une procédure contient, dans l'ordre :

- un identificateur donnant le nom de la procédure ;
- la liste des index des types des arguments, chacun d'eux étant précédé d'un mot-clé indiquant s'il s'agit :
  - d'un paramètre d'entrée (mot-clé `i:`), dont la procédure reçoit la valeur ;
  - d'un paramètre de sortie (mot-clé `o:`), dont la procédure définit la valeur ;
  - ou d'un paramètre d'entrée-sortie (mot-clé `io:`), dont la procédure reçoit la valeur avant de la redéfinir.
- éventuellement, les mots-clés `memorysafe:` ou `unsafe:` (cf. 2.4.5, page 26), en l'absence desquels la procédure est supposée *safe* (cf. 2.4.5, page 26).

♠ **OBJET-PROCÉDURE** ::=

• **Identificateur-***procédure* `(` [{ **PARAMÈTRE** `,` ... }] `)` [**Propriété-de-processus**] •

♠ **PARAMÈTRE** ::= • **PARAMÈTRE-ENTRÉE** •
• **PARAMÈTRE-SORTIE** •
• **PARAMÈTRE-ENTRÉE-SORTIE** •

♠ **PARAMÈTRE-ENTRÉE** ::= • `i:` **Index-***type* •

♠ **PARAMÈTRE-SORTIE** ::= • `o:` **Index-***type* •

♠ **PARAMÈTRE-ENTRÉE-SORTIE** ::= • `io:` **Index-***type* •

2.4.5 Propriétés sémantiques communes des processus

Quelle que soit leur forme syntaxique, les modules, nœuds, fonctions, procédures (appelés ici génériquement processus) jouissent de l'une des propriétés sémantiques suivantes :

safe : un processus est dit "*safe*" s'il est une fonction au sens mathématique du terme, sans mémoire (i.e., constante sur le temps) ; cette propriété est dénotée par l'attribut `safe:`, qui est l'attribut par défaut pour les fonctions et procédures ;

memorysafe : un processus est dit "*memorysafe*" s'il réalise une fonction des états initiaux, des trajectoires des entrées, des trajectoires des horloges de sortie dans les trajectoires des sorties ; cette propriété est dénotée par l'attribut `memorysafe:` ; pour ces processus le principe de substitution s'applique.

unsafe : un processus est dit "*unsafe*" dans tous les autres cas ; cette propriété est dénotée par l'attribut `unsafe:`.

Ces propriétés sont ordonnées comme suit :

- tout processus *safe* est *memorysafe* ;
- tout processus *memorysafe* est *unsafe*.

♠ **Propriété-de-processus** ::= • `safe:` • `memorysafe:` • `unsafe:` •

2.4.6 Blocs de données

Les blocs de données fournissent une description syntaxique d'objets externes au format (types, constantes, fonctions et procédures). Plusieurs entités peuvent importer de tels blocs pour atteindre les **objets** décrits. Un bloc de données peut lui-même importer d'autres blocs de données en en donnant les index externes dans sa liste d'importation. Cette liste ne peut pas comporter d'autres entités que des blocs de données.

Chaque partie est optionnelle, sauf l'indication de format et le nom.

♠ **BLOC-DE-DONNÉES** ::= • `data:` **INFOS-BLOC-DE-DONNÉES**
[TABLE-IMPORTATIONS]
[TABLE-TYPES]
[TABLE-CONSTANTES]
[TABLE-FONCTIONS]
[TABLE-PROCÉDURES]
[TABLE-PRAGMAS]
`enddata:` •

♠ **INFOS-BLOC-DE-DONNÉES** ::=

• **Format-dc Identificateur-***bloc-de-données* [`flat:`] •

♠ **Format-dc** ::= • `dc:` **Numéro-version** •

Exemple de bloc de données Cet exemple est présenté sur deux colonnes.

```
data: dc:0 ARRAY
   types: 2
     0: ELEM_TYPE;
     1: ARRAY_TYPE;
   end:

   constants: 2
     0: LINESIZE $2;
     1: COLUMNSIZE $2;
   end:
```

```
   functions: 7
     0: GET (1, $2, $2): 0 safe:;
     1: $eq (0, 0): $1;
     2: $ne (0, 0): $1;
     3: $cond ($1, 0, 0): 0;
     4: $eq (1, 1): $1;
     5: $ne (1, 1): $1;
     6: $cond ($1, 1, 1): 1;
   end:

   procedures: 4
     0: $assign (o: 0, i: 0) memorysafe:;
     1: $assign (o: 1, i: 1) memorysafe:;
     2: SET (io: 1, i: $2, i: $2, i:0) memorysafe:;
     3: PROCGET (i: 1, i: $2, i: $2, o: 0) safe:;
   end:
enddata:
```

Chapitre 3

Les éléments communs aux formats impératifs IC et OC

Ce chapitre décrit la partie commune spécifique aux deux formats intermédiaires impératifs des langages synchrones.

3.1 Organisation des données IC et OC

3.1.1 Variables

La table des variables est identifiée par le mot-clé **variables:**. On rappelle que la syntaxe générique des tables est décrite en 2.2.3.

♠ **Nom-table-variables** ::= • **variables:** •

Cette table rassemble les variables déclarées dans le programme et les variables générées par le compilateur (statut et valeur des signaux, compteurs, etc.). Une entrée de cette table comporte deux champs, le second étant optionnel. Les informations données sur la variable considérée sont :

- l'index du type de la variable ;
- éventuellement, la valeur initiale de la variable précédée du mot-clé **value:**. Cette valeur est une expression de même type que la variable et dont la syntaxe est donnée en 2.2.5.

♠ **OBJET-VARIABLE** ::= • **Index-***type* [**VALEUR-INITIALE**] •

♠ **VALEUR-INITIALE** ::= • **value:** **EXPRESSION** •

3.1.2 Signaux

La table des signaux est identifiée par le mot-clé **signals:**. On rappelle que la syntaxe générique des tables est décrite en 2.2.3.

♠ **Nom-table-signaux** ::= • **signals:** •

Il existe deux sortes de signaux : les signaux d'interface entre le programme et son environnement d'une part, et les signaux internes au programme d'autre part. Les signaux d'interface sont de quatre types : entrée, terminaison normale de tâche, sortie ou entrée/sortie.
Voici les divers objets pouvant être associés à un signal :

- à chaque signal d'interface est associé un nom identifiant ce signal à l'extérieur du module ;
- à chaque signal valué est associée une variable conservant la valeur courante du signal ;
- à chaque signal valué qui a été déclaré émissible plusieurs fois dans le même instant est associée une fonction réalisant la combinaison de deux valeurs ; cette fonction doit prendre deux arguments du type de la valeur du signal et rendre un résultat de même type ; l'opérateur associé à cette fonction doit être associatif et commutatif.

Une entrée de la table des signaux rassemble, selon le type et la valeur du signal, les références aux objets listés plus haut. Une entrée de la table comporte donc successivement :

1. une **NATURE-DE-SIGNAL** ;
2. un **CANAL-DE-SIGNAL** ;
3. un booléen optionnel (**BOOL-DE-SIGNAL**).

Nous détaillons ci-dessous chacune de ces trois parties :

- **NATURE-DE-SIGNAL** représente les divers types de signaux :
  - pour les signaux d'entrée, nous trouvons le mot-clé **input:** suivi du nom du signal ;
  - pour les signaux de terminaison normale de tâche, nous trouvons le mot-clé **return:** suivi du nom du signal ; ces signaux sont une classe particulière des signaux d'entrée ;
  - pour les signaux de sortie, nous trouvons le mot-clé **output:** suivi du nom du signal ;
  - pour les signaux d'entrée-sortie, nous trouvons le mot-clé **inputoutput:** suivi du nom du signal ;
  - pour les signaux locaux, nous trouvons le mot-clé **local:** uniquement ;
- **CANAL-DE-SIGNAL** représente la valeur transportée par le signal. Un canal est composé de deux ou trois champs :
  - le premier champ indique le type du signal ;

- si le signal n'est pas de type **$pure**, cette information est suivie de l'index de la variable contenant la valeur ; si le signal est de type **$pure**, ce second champ est omis ;
- le deuxième champ indique si le signal peut être émis plusieurs fois dans le même instant : ce champ est soit **single:** si le signal ne peut être émis qu'une seule fois, soit **multiple:** suivi de l'index de la fonction de combinaison des valeurs.

Attention : Un signal **return:** ne peut pas être **multiple:**.

- **BOOL-DE-SIGNAL** est constitué du mot-clé **bool:** suivi d'un index de variable booléenne. Ce champ existe :
  - pour les signaux de nature **input:**, **return:** et **inputoutput:**. Il est utilisé pour tester la présence du signal.
  - pour les signaux de canal **multiple:**. Il est utilisé pour savoir si le signal a déjà été émis dans l'instant et appliquer alors la fonction de combinaison lors des émissions suivantes.

♠ **OBJET-SIGNAL** ::= • **NATURE-DE-SIGNAL CANAL-DE-SIGNAL [BOOL-DE-SIGNAL]** •

♠ **NATURE-DE-SIGNAL** ::= • **input:** **Identificateur-***signal* •
• **return:** **Identificateur-***signal* •
• **output:** **Identificateur-***signal* •
• **inputoutput:** **Identificateur-***signal* •
• **local:** •

♠ **CANAL-DE-SIGNAL** ::=
• **Index-***type* **[VARIABLE-ASSOCIÉE] CARDINALITÉ** •

♠ **VARIABLE-ASSOCIÉE** ::= • **value:** **Index-***variable* •

♠ **CARDINALITÉ** ::= • **single:** •
• **multiple:** **Index-***fonction* •

♠ **BOOL-DE-SIGNAL** ::= • **bool:** **Index-***variable* •

3.1.3 Expressions spécifiques IC et OC

Les expressions spécifiques aux formats impératifs sont les expressions sur les signaux et les variables, qui complètent ainsi les expressions décrites en 2.2.5 :

- le statut de présence d'un signal est représenté par le caractère **!** suivi de l'index du signal ;

- la valeur courante d'un signal est représentée par le caractère ? suivi de l'index du signal ;
- une variable est simplement référencée par son index.

♠ **Terme-IC-OC** ::= • **Présence-signal** •
• **Valeur-signal** •
• **Index-***variable* •

♠ **Présence-signal** ::= • ! **Index-***signal* •

♠ **Valeur-signal** ::= • ? **Index-***signal* •

Dans un module IC (cf. 4.2, page 39) et dans un module OC (cf. 6.2, page 83) :

- les index de signaux et de variables apparaissant dans une expression sont des index locaux ;
- les index de constantes apparaissant dans une expression peuvent être soit des index de constantes prédéfinies communes, soit des index composés désignant des constantes de blocs de données importés ;
- les index de fonctions apparaissant dans une expression peuvent être soit des index de fonctions prédéfinies communes, soit des index composés désignant des fonctions de blocs de données importés ou prédéfinis.

3.1.4 Types spécifiques IC et OC

Il n'y a pas de types spécifiques aux formats impératifs.

3.1.5 Fonctions et procédures

Les paramètres effectifs **i:** des procédures et des fonctions sont des expressions sur variables et signaux ; les paramètres **io:**, **o:** sont des variables.

3.1.6 Fonctions prédéfinies spécifiques IC et OC

Les formats impératifs utilisent les trois fonctions prédéfinies supplémentaires suivantes :

- `left_and` et `left_or` sont les deux fonctions booléennes classiques *et* et *ou* qui n'évaluent pas systématiquement leurs deux arguments,
- la fonction `dsz` décrémente la variable passée en argument et renvoie *vrai* (constante `$2`) si et seulement si la nouvelle valeur de la variable est `0`.

```
functions: 3
  -- boolean
  $I0: $left_and ($1, $1): $1;
  $I1: $left_or  ($1, $1): $1;
  -- integer
  $I2: $dsz ($2): $1 unsafe:;
end:
```

3.1.7 Procédures prédéfinies spécifiques IC et OC

Rappelons que pour un type μ quelconque, peut exister la procédure d'affectation de nom prédéfini `$assign`.

La table 3.1 donne les procédures prédéfinies, ce sont les procédures d'affectation associées aux types prédéfinis.

```
procedures: 5
  $I0: $assign (o: $1, i: $1);
  $I1: $assign (o: $2, i: $2);
  $I2: $assign (o: $3, i: $3);
  $I3: $assign (o: $4, i: $4);
  $I4: $assign (o: $5, i: $5);
end:
```

Tableau 3.1 : Les procédures prédéfinies

3.1.8 Exemple

Les déclarations ESTEREL suivantes :

```
type USER_TYPE;
function PLUS(integer, integer) : integer;
output O : combine integer with PLUS;
input T : USER_TYPE,
      S;
```

produisent dans un même paquet par exemple :

- un bloc de données de numéro 3 contenant les tables suivantes :

```
types: 1
   0: USER_TYPE;
end:
```

```
functions: 4
   0: PLUS ($2,$2):$2;
   1: $eq (0,0):$1;
   2: $ne (0,0):$1;
   3: $cond ($1,0,0):0;
end:
```

```
procedures: 1
   0: $assign (o: 0, i: 0);
end:
```

- un module contenant une table des importations contenant (par exemple) en position 1 l'index du bloc de données 3 ci-dessus et les tables suivantes :

```
variables: 5
   0: $2;
   1: $1;
   2: 1.0;
   3: $1;
   4: $1;
end:
```

```
signals: 3
   0: output: O $2 value: 0 multiple: 1.0 bool: 1;
   1: input: T 1.0 value: 2 single: bool: 3;
   2: input: S $0           single: bool: 4;
end:
```

3.2 Tâches

Les tâches sont des processus externes qui sont exécutés par le système d'exploitation en parallèle avec le programme synchrone ; ce dernier contrôle l'exécution des processus externes par des actions spécifiques (`start`, `kill`, `suspend` et `resume`).

3.2.1 Description d'une tâche

La table des tâches est identifiée par le mot-clé **tasks:**. On rappelle que la syntaxe générique des tables est décrite en 2.2.3.

♠ **Nom-table-tâches** ::= • **tasks:** •

La description d'une tâche contient, dans l'ordre :

- un identificateur donnant le nom de la tâche ;
- la liste des index des types des arguments, chacun d'eux étant précédé d'un mot-clé indiquant s'il s'agit :
  - d'un paramètre d'entrée (mot-clé **i:**), que la tâche peut lire mais n'a pas le droit de modifier ;
  - d'un paramètre de sortie (mot-clé **o:**), auquel la tâche peut affecter une valeur mais qu'elle ne peut pas lire ;
  - ou d'un paramètre d'entrée-sortie (mot-clé **io:**), que la tâche peut lire et modifier.

La syntaxe est donc la suivante :

♠ **OBJET-TÂCHE** ::=

• **Identificateur-***tâche* **(** [{ **PARAMÈTRE** **,** ... }] **)**
[**Propriété-de-processus**] •

3.2.2 Description d'une occurrence de tâche

La table des occurrences de tâches est identifiée par le mot-clé **execs:**. On rappelle que la syntaxe générique des tables est décrite en 2.2.3.

♠ **Nom-table-appels-tâches** ::= • **execs:** •

La table des occurrences de tâches rassemble les divers appels de tâches qui apparaissent dans le code source. À un appel de tâche source correspond une et une seule entrée dans la table des occurrences de tâches. Il peut y avoir plusieurs occurrences de tâches pour une même tâche de la même manière qu'une procédure peut être appelée plusieurs fois.

Une occurrence de tâche rassemble les informations nécessaires pour contrôler l'exécution d'une tâche. On trouve donc, dans l'ordre :

- l'index de la tâche associée dans la table des tâches,
- l'index du signal de terminaison normale (signal **return:**) de la tâche (cf. 3.1.2, page 29),
- une liste d'expressions entre parenthèses et séparées par des virgules qui sont les arguments de la tâche dans cet appel. À tout paramètre formel déclaré avec le préfixe **o:** ou **io:** doit correspondre un paramètre effectif qui est un index de variable.

♠ **OBJET-APPEL-TÂCHE** ::=

• **Index-*tâche*** **Index-*signal*** **(** [{ **EXPRESSION** **,** ... }] **)** •

3.2.3 Exemple

Le code ESTEREL ci-dessous : donne les tables ci-dessous :

```
type POSITION;

task MOVE_ARM(POSITION)(),
     CLOSE_HAND()(),
     OPEN_HAND()();

...

   exec OPEN_HAND()();
   exec MOVE_ARM(POS)();
   exec CLOSE_HAND()();
...
   exec OPEN_HAND()();
```

```
tasks: 3
   0: MOVE_ARM (io: 0);
   1: CLOSE_HAND ();
   2: OPEN_HAND ();
end:

execs: 4
   0: 2 0 ();
   1: 0 1 (0);
   2: 1 2 ();

   3: 2 3 ();
end:
```

3.3 Actions

La table des actions est identifiée par le mot-clé **actions:**. On rappelle que la syntaxe générique des tables est décrite en 2.2.3.

♠ **Nom-table-actions** ::= • **actions:** •

Une description d'action est une liste non vide d'actions élémentaires (sans séparateur). Ces actions élémentaires peuvent être classées en 12 catégories :

- `act:` ⟨*index-d'action*⟩ exécute l'action d'index ⟨*index-d'action*⟩. *Les liaisons par les actions élémentaires* `act:` *ne doivent bien sûr pas créer de cycle.*
- `if:` ⟨*expression*⟩ teste l'expression booléenne (de type `$1`) ⟨*expression*⟩. Cette action élémentaire ne peut se trouver qu'à la fin d'une liste d'actions élémentaires car le choix de l'action suivante à exécuter dépend du résultat du test. Par exemple, "`if: $10(@3, 2)`" traduit "`if LIMIT_TIME < TIME`" si `LIMIT_TIME` est la constante 3 et `TIME` la variable 2 ; `if: !3` teste la présence du signal d'index 3.

- `call`: ⟨*index-de-procédure*⟩(⟨*liste-d'expressions*⟩) appelle la procédure référencée par ⟨*index-de-procédure*⟩ avec les arguments donnés dans la liste d'expressions. A tout paramètre formel déclaré avec le préfixe **o:** ou **io:** doit correspondre un paramètre effectif qui est un index de variable.

- `input`: ⟨*index-de-signal*⟩ réalise l'acquisition de l'extérieur de la présence et de la valeur du signal d'entrée d'index ⟨*index-de-signal*⟩.

- `output`: ⟨*index-de-signal*⟩ émet vers l'extérieur le signal de sortie d'index ⟨*index-de-signal*⟩.

- `reset`: ⟨*index-de-variable*⟩ remet la variable d'index ⟨*index-de-variable*⟩ à sa valeur initiale. Si la variable n'a pas de valeur initiale, elle est mise dans l'état *non initialisée*.

- `combine`: ⟨*index-de-signal*⟩⟨*expression*⟩ calcule la valeur d'un signal multiple. Si le signal n'a jamais été émis, la valeur du signal devient ⟨*expression*⟩. Si le signal a déjà été émis, il a une valeur ; sa valeur et l'⟨*expression*⟩ sont passées en argument à la fonction de combinaison du signal et le résultat devient la nouvelle valeur du signal. Rappelons que les fonctions de combinaison prennent deux arguments et rendent un résultat de même type : celui de la valeur du signal. Par exemple, en supposant que la constante 5 se nomme `C`, `combine: 3 $14(#3, @5)` met `C+3` comme valeur du signal 3 s'il n'a pas été émis. Si le signal a déjà été émis, sa nouvelle valeur est le résultat de l'application de la fonction de combinaison à son ancienne valeur et à `C+3`.

- `start`: ⟨*index-d'exec*⟩ lance l'exécution d'une nouvelle occurrence de tâche donnée par ⟨*index-d'exec*⟩.

- `kill`: ⟨*index-d'exec*⟩ tue l'occurrence de tâche donnée par ⟨*index-d'exec*⟩.

- `return`: ⟨*index-d'exec*⟩ écrit les paramètres par référence (préfixés par **o:** ou **io:**) de l'occurrence de tâche donnée par ⟨*index-d'exec*⟩.

- `suspend`: ⟨*index-d'exec*⟩ suspend l'exécution de l'occurrence de tâche donnée par ⟨*index-d'exec*⟩.

- `resume`: ⟨*index-d'exec*⟩ reprend l'exécution de l'occurrence de tâche donnée par ⟨*index-d'exec*⟩.

Les actions elles-mêmes sont classées en deux catégories. On distingue en effet les actions "pures" des actions de test :

- une action pure est une action ne contenant que des actions élémentaires de type **input:**, **output:**, **call:**, **reset:**, **combine:**, **start:**, **kill:**, **return:**, **suspend:** ou **resume:** ou des actions élémentaires de type **act:** référençant des actions pures ;

- une action de test est une action pure suivie d'une action élémentaire de type **if:** ou d'une action élémentaire de type **act:** référençant une action de test.

♠ **OBJET-ACTION** ::= • { **ACTION-ÉLÉMENTAIRE** ... } •

♠ **ACTION-ÉLÉMENTAIRE** ::=

- **act:** Index-*action* •
- **if:** **EXPRESSION** •
- **call:** Index-*procédure* **(** [{ **EXPRESSION** **,** ... }] **)** •
- **input:** Index-*signal* •
- **output:** Index-*signal* •
- **reset:** Index-*variable* •
- **combine:** Index-*signal* **EXPRESSION** •
- **start:** Index-*exec* •
- **kill:** Index-*exec* •
- **return:** Index-*exec* •
- **suspend:** Index-*exec* •
- **resume:** Index-*exec* •

À titre d'exemple, voici la table des actions d'un petit programme ESTEREL :

```
actions: 26
0: if:!0;
1: if:!1;
2: if:!2;
3: if:!3;
4: output:4;
5: output:5
    call:$1 (11) (@0)
    call:$1 (10) (#0);
6: output:6;
7: output:7;
8: output:8
    act:10;
9: output:9;
10: output:10;
11: output:11;
12: call:$1 (4) (#0)
    act:25;
13: call:$1 (4) (10)
    call:$1 (5) (10)
    call:$1 (13) ($14 (13,5))
    call:$1 (14) ($14 (14,#1))
    call:$1 (6) ($18 (13,14))
    act:25;
14: call:$1 (4) (6)
    act:4;
15: call:$1 (8) (@0);
16: call:$1 (9) (0 ());
17: call:$1 (10) ($14 (10,#1));
18: call:$1 (12) (@2);
19: call:$1 (13) (#0)
    call:$1 (14) (#0)
    call:$1 (7) (@1)
    if:$12 (7,#0);
20: if:$19 (7);
21: if:$19 (8);
22: if:$19 (9);
23: if:$19 (11);
24: if:$19 (12);
25: act:4
    act:6;
end:
```

3.4 Instances

La table des instances est identifiée par le mot-clé **instances:**. On rappelle que la syntaxe générique des tables est décrite en 2.2.3.

Cette table est utilisée pour mettre en relation le code généré et le programme source correspondant. Le module auquel appartient cette table est appelé *module racine*. Dans

la table des instances, on trouve une entrée pour chaque instance de sous-module utilisé (par l'instruction Run) ainsi que pour le module racine. L'index du module racine est donné par l'information supplémentaire **root:**.

♠ **Nom-table-instances** ::= • **instances:** •

♠ **INFOS-TABLE-INSTANCES** ::= • **root:** **Index-***instance* **;** •

Chaque entrée de la table des instances contient :

- un identificateur donnant le nom du module ;
- l'index du module (appelé *module père*) utilisant le module correspondant à cette entrée. Pour le module racine, on utilise ici l'index indéfini **–**.

♠ **OBJET-INSTANCE** ::= • **Identificateur-***module* **Index-***module-père* •

3.5 Relations

La table des relations est identifiées par le mot-clé **relations:**. On rappelle que la syntaxe générique des tables est décrite en 2.2.3.

♠ **Nom-table-relations** ::= • **relations:** •

Cette table rassemble les relations entre signaux données dans le programme source. Les différentes informations sont :

- une liste d'index de signaux donnant les signaux ne pouvant pas être présents simultanément dans un même instant ;
- deux index de signaux lorsque la présence du premier signal implique celle du deuxième.

♠ **OBJET-RELATION** ::= • **EXCLUSION** • **IMPLICATION** •

♠ **EXCLUSION** ::= • **(** **Index-***signal* **,** { **Index-***signal* **,** ... } **)** •

♠ **IMPLICATION** ::= • **Index-***signal* **Index-***signal* •

Chapitre 4

Le format impératif parallèle : IC

4.1 Introduction

Le format IC est une représentation impérative et paralèlle d'un programme synchrone. Un code IC se compose de deux parties :

- une partie déclarative définissant des objets comme les signaux, les variables, ...
- une partie impérative codant le comportement du programme.

Structure d'un code IC

Un code IC est constitué d'une liste de déclarations de deux sortes d'entités : des *blocs de données* et des *modules*.

L'entité de base d'un code IC s'appelle un *module*. Un module décrit dans le format IC se compose de tables. Ces tables sont de deux types :

- des tables de données (signaux, variables, tâches, ...)
- une table d'instructions constituant le corps du module et utilisant les objets définis dans les autres tables.

4.2 Modules

Un module IC définit le comportement et l'interface d'un programme synchrone. Un module décrit dans le format IC consiste en :

- le mot-clé **module:** suivi du format IC puis du nom du module ;
- les attributs optionnels du module parmi lesquels l'attribut `linked:` indique que le module ne contient aucune instruction **Run:** ;
- les tables optionnelles communes aux formats impératifs ;
- la table des instructions ;
- le mot-clé **endmodule:**.

♠ MODULE-IC ::= • **module:** INFOS-MODULE-IC
PARTIE-COMMUNE-IMPÉRATIVE
TABLE-INSTRUCTIONS
endmodule: •

♠ INFOS-MODULE-IC ::= • **Format-ic Identificateur-***module*
[**flat:**] [**linked:**]
[**Propriété-de-processus**] •

♠ PARTIE-COMMUNE-IMPÉRATIVE ::= • [**TABLE-IMPORTATIONS**]
[**TABLE-INSTANCES**]
[**TABLE-VARIABLES**]
[**TABLE-SIGNAUX**]
[**TABLE-RELATIONS**]
[**TABLE-TÂCHES**]
[**TABLE-APPELS-TÂCHES**]
[**TABLE-ACTIONS**]
[**TABLE-PRAGMAS**] •

♠ Format-ic ::= • **ic:** **Numéro-version** •

4.3 Instructions

La table des instructions est identifiée par le mot-clé **instructions:**. On rappelle que la syntaxe générique des tables est décrite en 2.2.3.

♠ Nom-table-instructions ::= • **instructions:** •

Les informations spécifiques à la table des instructions sont les suivantes :

- l'index de l'instruction de départ précédé du mot-clé **startpoint:** ; cette instruction doit être une instruction de contrôle (cf ci-dessous) ;
- le nombre d'instructions `Goloop` précédé du mot-clé **goloops:** ;
- le niveau de sortie maximal des instructions `Exit` précédé du mot-clé **exitlevels:**.

♠ INFOS-TABLE-INSTRUCTIONS ::= • **startpoint:** **Index-***instruction* **;**
goloops: **Cst-entière** **;**
exitlevels: **Cst-entière** **;** •

On distingue :

- les *instructions de contrôle,* dont l'*arité* se définit comme le nombre de ses continuations :
  - `Exit`, `Halt`, `Watchdog`, `Stay` et `Return` sont d'arité 0,
  - `Action`, `Reset`, `Emit`, `Access`, `Goto`, `Goloop` et `Run` sont d'arité 1,

- `Test` est d'arité 2,
- `Fork` est d'arité variable supérieure à 0 ;

- les *instructions de reconstruction* :
  - `Halt`, `Watchdog`, `Stay` et `Return` qui sont également des instructions de contrôle (d'arité 0),
  - `Parallel`.

♠ **OBJET-INSTRUCTION** ::=

- **Action:** [**Index-***action*] (**Index-***instruction*) •
- **Test:** [**Index-***action*]
  (**Index-***instruction* , **Index-***instruction*) •
- **Reset:** [**Index-***signal*] (**Index-***instruction*) •
- **Emit:** [**Index-***signal*] (**Index-***instruction*) •
- **Access:** [**Index-***signal*] (**Index-***instruction*) •
- **Goto:** (**Index-***instruction*) •
- **Goloop:** (**Index-***instruction*) •
- **Fork:** { **Index-***instruction* } ({ **Index-***instruction* , ... }) •
- **Parallel:** ({ **Index-***instruction* , ... }) < **Index-***instruction* > •
- **Exit:** { **Index-***instruction* } **Cst-entière** •
- **Halt:** < **Index-***instruction* >
  [{ { **Index-***appel-tâche* , ... } }]**Cst-entière** •
- **Watchdog:** { **Index-***instruction* } < **Index-***instruction* > •
- **Stay:** < **Index-***instruction* > •
- **Run:** **Identificateur-***module* [**RENOMMAGES-IC**]
  < **Index-***instruction* > (**Index-***instruction*) •
- **Return:** **Cst-entière** •

L'instruction `Action: [a] (c)`

effectue l'action d'index a et passe le contrôle à l'instruction d'index c.

L'entier a doit être l'index d'une action pure dans la table des actions (cf. 3.3, page 35). L'entier c doit être l'index d'une instruction de contrôle dans la table des instructions.

L'instruction `Test: [a] (c1,c2)`

effectue l'action de test d'index a et passe le contrôle à l'instruction d'index c_1 si a rend vrai ou à l'instruction d'index c_2 si a rend faux.

L'entier a doit être l'index d'une action de test dans la table des actions (cf. 3.3, page 35). Les entiers c_1 et c_2 doivent être des index d'instructions de contrôle dans la table des instructions.

L'instruction `Reset`: [s] (c)

déclare que le signal d'index s doit être remis à l'état *non émis* et que sa valeur doit être remise à *indéfini* s'il est valué, puis passe le contrôle à l'instruction d'index c.

L'entier s doit être l'index d'un signal dans la table des signaux. L'entier c doit être l'index d'une instruction de contrôle dans la table des instructions.

Une instruction `Reset` est associée à chaque déclaration de signal local.

L'instruction `Emit`: [s] (c)

émet le signal d'index s et passe le contrôle à l'instruction d'index c.

L'entier s doit être l'index d'un signal dans la table des signaux. L'entier c doit être l'index d'une instruction de contrôle dans la table des instructions.

Si le signal est valué, le calcul de sa nouvelle valeur n'est pas effectué par l'instruction `Emit` ; il doit être effectué par une instruction `Action` générée séparément qui affecte la valeur à la variable du signal ou la combine à la valeur courante en cas de signal multiple.

L'émission vers l'extérieur d'un signal de sortie n'est pas effectuée par l'instruction `Emit`. Elle doit être réalisée par l'appel d'une action `Output`, appel qui ne doit avoir lieu que lorsqu'on est sûr qu'aucune instruction `Emit` de ce signal ne pourra plus être exécutée. Dans le code IC, il n'y a pas d'instruction explicite pour appeler cette action `Output` ; son appel est à la charge du mécanisme d'interprétation, qui pourra par exemple l'appeler à la fin de la réaction si le signal a été émis au moins une fois. Dans le code OC, les appels à l'action `Output` sont explicités et doivent donc être produits par le compilateur IC-OC.

L'instruction `Access`: [s] (c)

gère la synchronisation des émissions et des réceptions de signaux ; le contrôle reste bloqué tant qu'une émission du signal référencé par une instruction `Emit` reste possible ; lorsque le signal ne peut plus être émis, le contrôle est passé à l'instruction d'index c.

L'entier s doit être l'index d'un signal dans la table des signaux. L'entier c doit être l'index d'une instruction de contrôle dans la table des instructions.

L'instruction `Goto`: (c)

passe le contrôle à l'instruction d'index c.

L'entier c doit être l'index d'une instruction de contrôle dans la table des instructions.

Une instruction `Goto` remplace l'instruction `Return` d'un module instancié (cf. 4.3, page 44).

L'instruction `Goloop`: (c)

signale la fin du corps d'une boucle et passe le contrôle à l'instruction d'index c ; cette instruction permet la détection des boucles instantanées.

L'entier c doit être l'index d'une instruction de contrôle dans la table des instructions.

L'instruction `Fork`: $\{p\}$ $(c_1, c_2, \ldots, c_n)$

distribue instantanément le contrôle aux n instructions d'index $c_1, c_2, \ldots, c_n$; l'exécution de ces instructions est synchronisée par le `Parallel` d'index p.

L'entier p doit être l'index d'une instruction `Parallel` dans la table des instructions. On doit avoir $n > 0$, et les entiers $c_1, c_2, \ldots, c_n$ doivent être des index d'instructions de contrôle dans la table des instructions.

L'instruction `Parallel`: $(c_0, c_2, \ldots, c_n)$ <r>

synchronise à chaque instant l'exécution des branches lancées par un `Fork` associé.

L'entier r doit être un index d'instruction de reconstruction `Parallel`, `Watchdog`, `Stay` ou `Return` dans la table des instructions. On doit avoir $n = 0$, ou $n \geq 2$, et les entiers $c_0, c_2, \ldots, c_n$ formant la *liste des continuations* doivent être des index d'instructions de contrôle dans la table des instructions (attention : il n'y a pas d'index 1 comme expliqué plus loin). L'index de l'instruction `Parallel` doit être référencé par une et une seule instruction `Fork`.

À chaque instant, chaque branche encore active envoie un *code de terminaison* par l'exécution d'une instruction `Halt` ou `Exit` ; l'instruction `Halt` envoie le code 1, et une instruction "`Exit` $\{p\}$ l" envoie le code l. L'instruction `Parallel` calcule le maximum m des codes de terminaisons des branches actives. Dans la liste des continuations d'une instruction `Parallel`, la première continuation correspond au niveau 0, la deuxième au niveau 2, la troisième au niveau 3, etc. Le niveau 1 étant traité à l'aide de l'arbre de reconstruction et donc du champ <r>, c'est pour cette raison qu'il n'y a pas d'instruction c_1 dans la liste des continuations.

- Si $m = 0$, toutes les branches sont terminées. Dans ce cas, l'instruction `Parallel` se termine également et passe le contrôle à l'instruction d'index c_0.

- Si $m = 1$, certaines branches ont pu se terminer, au moins une branche a exécuté une instruction `Halt`, et aucune branche n'a effectué d'instruction `Exit` avec $l > 1$. Dans ce cas, l'instruction `Parallel` n'est pas encore terminée, et elle envoie le code de terminaison 1 à son parallèle père éventuel dans l'arbre de reconstruction ; elle se comporte donc exactement comme un `Halt`.

- Si $m > 1$, une des branches a demandé la sortie immédiate du parallèle. Toutes les branches sont alors tuées.

  - Si $m \leq n$, la sortie est traitée localement en envoyant le contrôle à l'instruction d'index c_m.

  - Si $m > n$, la sortie n'est pas traitée localement. On envoie le niveau de sortie $m - n + 1$ au `Parallel` ou `Return` père dans l'arbre de reconstruction, le décalage provenant du fait que le niveau 1 ne correspond pas à une continuation explicite.

L'instruction `Exit`: $\{p\}$ l

signale au parallèle d'index p une terminaison de niveau l.

L'entier p doit être l'index d'une instruction `Parallel`. L'entier l doit être différent de 1.

L'instruction `Halt: <r> i` ou `Halt: <r> {e} i`

bloque le contrôle et envoie le code de terminaison 1 à son `Parallel` ou `Return` père dans l'arbre de reconstruction.

L'entier r doit être l'index d'une instruction de reconstruction `Parallel`, `Watchdog`, `Stay` ou `Return` dans la table des instructions. S'il est présent, l'entier e doit être l'index d'un `Exec` dans la table des execs. *Un tel entier e doit apparaître au plus une fois dans la table des instructions.*

L'entier $i \geq 0$ est appelé le *rang* du `Halt`. Les rangs des `Halt` et le rang du `Return` doivent être tous distincts et énumérer l'ensemble $\{0, 1, ..., n\}$ s'il y a n `Halt` dans la table des instructions. Le champ *rang-de-halt* est un numéro unique identifiant le `Halt` et utile pour coder les états sous forme d'ensemble d'entiers implémentés en chaînes de bits (un état est simplement un ensemble de `Halt`. Dans le cas d'un `Halt` associé à un `exec`, le numéro d'`exec` permet de gérer la tâche externe référencée par l'`exec` comme expliqué en 3.2.2.

L'instruction `Watchdog: {p} <r>`

contrôle l'exécution de la partie de l'arbre de reconstruction qu'il domine.

L'entier p doit être l'index d'une instruction de contrôle dans la table des instructions. L'entier r doit être l'index d'une instruction de reconstruction `Parallel`, `Watchdog`, `Stay` ou `Return` dans la table des instructions.

L'instruction `Stay: <r>`

gèle dans leur état courant tous les `Halt` qu'elle domine dans l'arbre de reconstruction.

L'entier r doit être l'index d'une instruction de reconstruction `Parallel`, `Watchdog`, `Stay` ou `Return` dans la table des instructions.

L'instruction `Run:` *nom-de-module* `[`*renommages*`] <r> (c)`

exécute le module d'identificateur *nom-de-module*; si le module termine, le contrôle passe à l'instruction d'index c.

Le champ *renommages* donne les informations suivantes dans cet ordre :

- une liste de renommages de types séparés par des virgules et suivie de ';'.
- une liste de renommages de constantes séparés par des virgules et suivie de ';'.
- une liste de renommages de fonctions séparés par des virgules et suivie de ';'.
- une liste de renommages de procédures séparés par des virgules et suivie de ';'.
- une liste de renommages de tâches séparés par des virgules et suivie de ';'.
- une liste de renommages de signaux séparés par des virgules et suivie de ';'.
- l'index du dernier signal déclaré au niveau de l'instruction d'expansion d'un module source.

Chaque renommage a la forme suivante :

$$nom - d'objet/index - d'objet$$

où *nom-d'objet* est le nom d'un objet dans le module instancié. Cet objet est renommé par l'objet référencé par *index-d'objet* dans le module courant.

♠ **RENOMMAGES-IC** ::= • **RENOMMAGE-TYPES-IC** [;]
RENOMMAGE-CONSTANTES-IC [;]
RENOMMAGE-FONCTIONS-IC [;]
RENOMMAGE-PROCÉDURES-IC [;]
RENOMMAGE-TÂCHES-IC [;]
RENOMMAGE-SIGNAUX-IC [;]
Index-*dernier-signal* •

♠ **RENOMMAGE-TYPES-IC** ::=
• { **RENOMMAGE-TYPE-IC** [,] ... } •

♠ **RENOMMAGE-CONSTANTES-IC** ::=
• { **RENOMMAGE-CONSTANTE-IC** [,] ... } •

♠ **RENOMMAGE-FONCTIONS-IC** ::=
• { **RENOMMAGE-FONCTION-IC** [,] ... } •

♠ **RENOMMAGE-PROCÉDURES-IC** ::=
• { **RENOMMAGE-PROCÉDURE-IC** [,] ... } •

♠ **RENOMMAGE-TÂCHES-IC** ::=
• { **RENOMMAGE-TÂCHE-IC** [,] ... } •

♠ **RENOMMAGE-SIGNAUX-IC** ::=
• { **RENOMMAGE-SIGNAL-IC** [,] ... } •

♠ **RENOMMAGE-TYPE-IC** ::= • **Identificateur-***type* [/] **Index-***type* •

♠ **RENOMMAGE-CONSTANTE-IC** ::=
• **Identificateur-***constante* [/] **Index-***constante* •

♠ **RENOMMAGE-FONCTION-IC** ::=
• **Identificateur-***fonction* [/] **Index-***fonction* •

♠ **RENOMMAGE-PROCÉDURE-IC** ::=
• **Identificateur-***procédure* [/] **Index-***procédure* •

♠ **RENOMMAGE-TÂCHE-IC** ::= • **Identificateur-***tâche* [/] **Index-***tâche* •

♠ RENOMMAGE-SIGNAL-IC ::= • Identificateur-*signal* [/] Index-*signal* •

L'instruction `Return` : i

indique la terminaison globale du module.

L'entier i, appelé le *rang*, doit être positif ou nul.

Elle possède un rang comme un `Halt` car elle intervient dans les codages d'états. Intuitivement, l'instruction `Return` peut être traitée de deux façons :

- Lorsqu'on compile un module complet, l'instruction `Return` est considérée comme une instruction `Halt`.
- Lorsqu'on instancie un module dans un autre module, l'instruction `Return` est remplacée par une instruction `Goto` pointant sur la continuation qui est indiquée dans l'instruction d'instanciation `Run`.

4.4 Correction des programmes IC

4.4.1 Arbre de reconstruction

Toute les instructions de reconstruction sauf `Return` ont un champ de reconstruction <*r*>. La loi de correction de la reconstruction est la suivante :

> *Il doit y avoir une et une seule instruction* `Return` *dans une table. Le champ de reconstruction* <r> *de toute instruction de reconstruction sauf* `Return` *doit pointer sur une autre instruction de reconstruction. Si l'on construit un graphe dont les sommets sont les index des instructions de reconstruction et ayant un arc de i vers j si le champ de reconstruction de j est* <*i*>*, alors ce graphe doit être un arbre ayant pour racine l'index de l'instruction* `Return`*, pour nœuds les index de toutes les instructions* `Watchdog`*,* `Parallel`*, et* `Stay`*, et pour feuilles les index de toutes les instructions* `Halt`*.*

Ceci définit l'*arbre de reconstruction* d'un programme IC.

4.4.2 Correction des niveaux d'exit

Dans une instruction "`Exit` $\{p\}$ l", l'entier p doit doit être l'index d'une instruction `Parallel` et l'entier l, qui représente un niveau de sortie, doit être différent de 1 ; rappelons que le niveau de sortie 1 n'est pas utilisé car il correspond au niveau de sortie implicite de l'instruction `Halt`.

Le niveau de sortie ne peut être quelconque car il doit correspondre à une continuation d'un parallèle englobant. Comme expliqué dans la description de l'instruction `Parallel`, cette continuation peut être directement traitée par le parallèle d'index p, ou elle peut être traitée par un parallèle père de p dans l'arbre de reconstruction. Soit n le nombre de continuations du parallèle d'index p. Le niveau de sortie l est correct s'il vérifie $l \leq n$ (dans ce cas, la sortie est directement traitée par le parallèle p). Si $l > n$, soit p' l'index du premier parallèle rencontré en remontant l'arbre de reconstruction depuis p, appelé parallèle père de p. Alors le niveau de sortie l est correct pour le `Parallel` d'index p si et seulement si le niveau de sortie $l - n + 1$ est correct pour le `Parallel` d'index p'.

4.4.3 Correction des continuations

Les champs de continuations des instructions IC ne peuvent pas être quelconques. En particulier, dans une branche de parallèle, il ne doit pas être possible qu'une continuation réfère à une instruction située hors du parallèle ou dans une autre de ses branches. Nous décrivons ici la correction des continuations d'une table d'instructions IC. Pour cela, nous définissons un parcours du graphe formé par ces instructions, en utilisant les champs de continuation c et les champs de reconstruction r. Ce parcours va marquer chaque instruction (atteignable) en y apposant une marque de la forme $M = (p, b)$ où p est l'index d'une instruction `Parallel` ou de l'instruction `Return`, et b est un numéro de branche de parallèle ou 0 pour l'instruction `Return`.

Au début, aucune instruction n'a de marque. Le parcours est défini par une fonction $Mark(i, M)$ où i est l'index de l'instruction courante à explorer et M est une marque. Cette fonction peut soit terminer normalement si le code IC est correct, soit déclarer le code IC incorrect et stopper immédiatement. La correction globale se teste en calculant $Mark(s, (r, 0))$ où s est l'index donné dans le champ `startpoint:` au début de la table des instructions et où r est l'index de l'instruction `Return`.

Voici comment on calcule $Mark(i, M)$:

- Si i est une instruction de contrôle unaire de la forme `Action`, `Reset`, `Emit`, `Access`, `Goto` ou `Run` ayant pour continuation c, il y a trois cas :
  - Si l'instruction i n'est pas encore marquée, on la marque avec la marque M, on appelle $Mark(c, M)$, puis on rend le contrôle à l'appelant.
  - Si l'instruction i est déjà marquée avec la marque M, on rend simplement le contrôle à l'appelant.
  - Si l'instruction i est déjà marquée avec une marque différente de la marque M, on déclare le code IC incorrect.
- Si i est une instruction `Goloop` de continuation c, il y a trois cas :
  - Si l'instruction i n'est pas encore marquée, on la marque avec M. Il y a deux sous-cas :
    - si c n'est pas déjà marquée avec la marque M, on déclare le code IC incorrect : un `Goloop` ne peut que revenir en arrière sur des instructions déjà explorées et de même marque ;
    - si c est déjà marquée avec la marque M, on rend le contrôle à l'appelant.
  - Si l'instruction i est déjà marquée avec la marque M, on rend simplement le contrôle à l'appelant.
  - Si l'instruction i est déjà marquée avec une marque différente de la marque M, on déclare le code IC incorrect.
- Si i est une instruction `Test` de continuations c_1 et c_2, il y a trois cas :
  - Si l'instruction n'est pas encore marquée, on la marque avec la marque M, on appelle $Mark(c_1, M)$ et $Mark(c_2, M)$, puis on rend le contrôle à l'appelant.

- Si l'instruction est déjà marquée avec la marque M, on rend simplement le contrôle à l'appelant.
- Si l'instruction est déjà marquée avec une marque différente de M, on déclare le code IC incorrect.

- Si i est une instruction **Exit** d'arguments p' et l, il y a deux cas :

  - Si la marque M est de la forme $M = (p, b)$ avec $p' \neq p$, on déclare le code IC incorrect. Intuitivement, le parallèle mentionné dans l'**Exit** n'est pas celui qu'on atteint par exécution du IC.
  - Sinon, on marque l'instruction avec la marque M et on rend simplement le contrôle à l'appelant (ceci fonctionne que l'instruction soit déjà marquée ou pas).

  Noter que le numéro de branche n'est pas testé pour les **Exit**. Donc, **Exit** est partageable entre les branches d'un **Parallel**, et c'est la seule instruction à avoir cette propriété.

- Si i est une instruction **Fork** de parallèle associé p' et de continuations $c_1, c_2, \ldots, c_n$, il y a trois cas :

  - Si l'instruction n'est pas encore marquée, on la marque avec la marque M, on appelle d'abord $Mark(c_1, (p', 1))$, $Mark(c_2, (p', 2))$, ..., $Mark(c_n, (p', n))$ pour vérifier le corps du nouveau couple **Fork-Parallel**, puis on appelle $Mark(p', M)$ pour vérifier les continuations du parallèle p', et enfin on rend le contrôle à l'appelant.
  - Si l'instruction est déjà marquée avec la marque M, on rend simplement le contrôle à l'appelant.
  - Si l'instruction est déjà marquée avec une marque différente de M, on déclare le code IC incorrect.

- Si i est une instruction **Parallel** de continuations $c_0, c_2, \ldots, c_n$ et de reconstruction r et si la marque M vérifie $M = (p, b)$, il y a deux cas car il y a deux façons de recevoir le contrôle pour *Mark* : soit par l'unique **Fork** associé au parallèle, ce qui ne peut se produire qu'une fois, soit depuis les instructions filles dans l'arbre de reconstruction lors de la remontée de cet arbre commandée par les instructions **Halt** (voir cette instruction plus loin) :

  - Si $p \neq i$, et si p n'est pas père de i dans l'arbre de reconstruction, alors on déclare le code IC incorrect. Si p est père de i, c'est qu'on est appelé depuis le **Fork**. Dans ce cas, on appelle $Mark(c_0, M), Mark(c_1, M), \ldots, Mark(c_n, M)$, et enfin $Mark(r, M)$. Il n'y a pas besoin de marquer l'instruction elle-même avec M puisque cet appel ne peut avoir lieu qu'une fois.
  - Si p est précisément l'index du **Parallel**, on rend simplement le contrôle à l'appelant. Ceci stoppe la remontée dans l'arbre de reconstruction décrite plus loin.

- Si i est une instruction `Halt` d'arguments r et j ou une instruction `Stay` d'argument r, il y a trois cas :
  - Si l'instruction n'est pas encore marquée, on marque l'instruction avec la marque M, on appelle $Mark(r, M)$, puis on rend le contrôle à l'appelant. Noter que l'appel $Mark(r, M)$ provoque une remontée de l'arbre de reconstruction initialisée par les `Halt` et remontant jusqu'au premier `Parallel` père, qui doit justement être celui mentionné dans M.
  - Si l'instruction est marquée avec M, on rend simplement le contrôle à l'appelant.
  - Si l'instruction est déjà marquée avec une marque différente de M, on déclare le code IC incorrect.
- Si i est une instruction `Watchdog` d'arguments p et r il y a trois cas :
  - Si l'instruction n'est pas marquée, on la marque avec M, on appelle $Mark(p, M)$, puis $Mark(r, M)$, et on rend le contrôle à l'appelant. Noter que l'appel $Mark(r, M)$ provoque une remontée de l'arbre de reconstruction.
  - Si l'instruction est marquée avec M, on rend simplement le contrôle à l'appelant.
  - Si l'instruction est déjà marquée avec une marque différente de M, on déclare le code IC incorrect.
- Si i est une instruction `Return`, si la marque est $M = (p, b)$ avec $p \neq i$, on déclare le code IC incorrect ; sinon, on rend le contrôle à l'appelant.

4.4.4 Exemples de calcul de correction de programmes IC

Exemple 1

Le programme ESTEREL suivant :

```
input I1, I2, I3;
output O, O1, O2;

trap T1 in
   [
      await I1
   ||
      trap T2 in
         do
            [
               await I1; exit T2
            ||
               await I2; exit T1
            ]
         watching I3
      handle T2 do
         emit O2
```

```
      end trap;
   ]
handle T1 do
   emit O1
end trap;
emit O
```

se traduit en IC de la façon suivante :

```
...
signals: 8
0: input: I1 $0 single: bool: 0 ;
1: input: I2 $0 single: bool: 1 ;
2: input: I3 $0 single: bool: 2 ;
3: output: O $0 single: ;
4: output: O1 $0 single: ;
5: output: O2 $0 single: ;
6: local: $0 single: ;
7: local: $0 single: ;
end:
...
statements: 27 startpoint: 1 ;
goloops: 0 ;
exitlevels: 3 ;
0: Return: 0 ;
1: Reset: [6] (2) ;
2: Fork: {3} (4, 8) ;
3: Parallel: (26, 25) <0> ;
4: Halt: <5> 1 ;
5: Watchdog: {6} <3> ;
6: Test: [0] (7,5) ;
7: Exit: {3} 0 ;
8: Reset: [7] (9) ;
9: Fork: {10} (11, 17) ;
10: Parallel: (7, 24) <22> ;
11: Halt: <12> 2 ;
12: Watchdog: {13} <10> ;
13: Test: [0] (14,12) ;
14: Emit: [7] (15) ;
15: Exit: {10} 2 ;
16: Exit: {10} 0 ;
17: Halt: <18> 3 ;
18: Watchdog: {19} <10> ;
19: Test: [1] (20,18) ;
20: Emit: [6] (21) ;
21: Exit: {10} 3 ;
22: Watchdog: {23} <3> ;
23: Test: [2] (7,22) ;
24: Emit: [5] (7) ;
25: Emit: [4] (26) ;
26: Emit: [3] (0) ;
end:
```

Nous allons maintenant vérifier la correction de ce programme IC à l'aide des critères de correction précédents.

Correction des valeurs auxiliaires Les informations utiles pour cette vérification sont les suivantes :

```
startpoint: 1 ;
goloops: 0 ;
exitlevels: 3 ;
...
1: Reset: [6] (2) ;
7: Exit: {3} 0 ;
15: Exit: {10} 2 ;
16: Exit: {10} 0 ;
21: Exit: {10} 3 ;
```

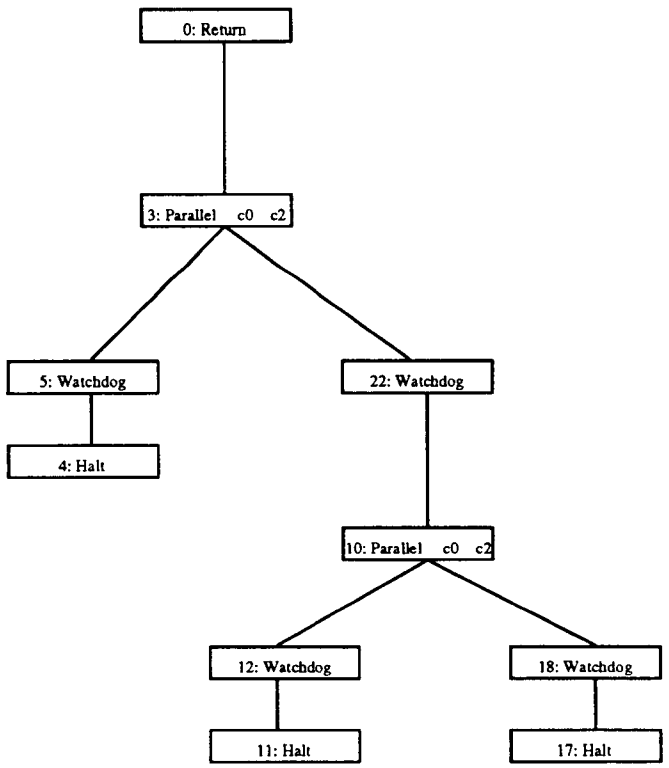
Le champ `startpoint` dénote une instruction de contrôle (**Reset**).
Il n'y a pas d'instruction `Goloop` dans le programme IC.

Le champ `exitlevels` donne le maximun 3 des niveaux de terminaison des quatre `Exit` du programme.

Correction de l'arbre de reconstruction Les instructions utiles sont :

```
0: Return: 0 ;
3: Parallel: (26, 25) <0> ;
5: Watchdog: {6} <3> ;
10: Parallel: (7, 24) <22> ;
11: Halt: <12> 2 ;
12: Watchdog: {13} <10> ;
17: Halt: <18> 3 ;
18: Watchdog: {19} <10> ;
22: Watchdog: {23} <3> ;
```

Il y a une et une seule instruction **Return**.
Le graphe construit à partir des informations de reconstruction donné par la figure :



est un arbre ayant pour racine l'index de l'instruction **Return**, pour nœuds les index de toutes les instructions **Watchdog**, **Parallel**, et **Stay** et pour feuilles les index de toutes les instructions **Halt**.

Correction des niveaux d'Exit Les seules informations utiles pour vérifier cette correction sont les suivantes :

```
3: Parallel: (26, 25) <0> ;
7: Exit: {3} 0 ;
```

```
10: Parallel: (7, 24) <22> ;
15: Exit: {10} 2 ;
16: Exit: {10} 0 ;
21: Exit: {10} 3 ;
22: Watchdog: {23} <3> ;
```

On vérifie bien que les niveaux de terminaison des `Exit` (`0, 2, 0, 3`) sont différents de 1.

Ensuite, les `Exit` font référence à deux instructions (3 et 10) qui sont bien des `Parallel`.

Finalement, nous vérifions pour chaque `Exit` que son niveau de terminaison est traité par un `Parallel` père :

- `7: Exit: {3} 0 ;`

  Le `Parallel` 3 traite le niveau de terminaison 0 par la continuation 26.

- `15: Exit: {10} 2 ;`

  Le `Parallel` 10 traite le niveau de terminaison 2 par la continuation 24.

- `16: Exit: {10} 0 ;`

  Le `Parallel` 10 traite le niveau de terminaison 0 par la continuation 7.

- `21: Exit: {10} 3 ;`

  Le `Parallel` 10 ne traite pas le niveau de terminaison 3. Il transmet donc à son `Parallel` père le niveau de terminaison $3 - 2 + 1 = 2$. Le `Parallel` père du `Parallel` 10 se trouve en remontant sur le `Watchdog` 22 et enfin sur le `Parallel` 3. Le `Parallel` 3 traite le niveau de terminaison 2 par la continuation 25.

Les niveaux d'`Exit` sont corrects.

Correction de continuations Nous appliquons la fonction *Mark*. Dans les tableaux qui suivent, les appels successifs à *Mark* sont séparés par une double ligne et sont réalisés en parallèle.

Instructions	M	Appels
`startpoint: 1` `0: Return: 0`		$Mark(1,(0,0))$

`startpoint: 1`
`0: Return: 0`
`1: Reset: [6] (2)`
	
	(0, 0)
		
		$Mark(2,(0,0))$

`0: Return: 0`
`1: Reset: [6] (2)`
`2: Fork: {3} (4, 8)`
	
	(0, 0)
	(0, 0)
		
		$Mark(4,(3,1))$ $Mark(8,(3,2))$
		$Mark(3,(0,0))$

`0: Return: 0`
`1: Reset: [6] (2)`
`2: Fork: {3} (4, 8)`
`3: Parallel: (26, 25) <0>`
 	
`4: Halt: <5> 1`
`8: Reset: [7] (9)`
	
	(0, 0)
	(0, 0)
	–
	
	(3, 1)
	(3, 2)
		
		0 est père de 3
		$Mark(26,(0,0))$ $Mark(25,(0,0)$
		$Mark(5,(3,1))$
		$Mark(9,(3,2))$

`0: Return: 0`
`1: Reset: [6] (2)`
`2: Fork: {3} (4, 8)`
`3: Parallel: (26, 25) <0>`
`4: Halt: <5> 1`
`5: Watchdog: 6 <3>`
`8: Reset: [7] (9)`
`9: Fork: {10} (11, 17)`
 	
`25: Emit: [4] (26)`
`26: Emit: [3] (0)`
	
	(0, 0)
	(0, 0)
	–
	(3, 1)
	(3, 1)
	(3, 2)
	(3, 2)
	
	(0, 0)
	(0, 0)
		
		$Mark(6,(3,1))$ $Mark(3,(3,1))$
		
		$Mark(11,(10,1))$ $Mark(17,(10,2))$
		$Mark(10,(3,2))$
		$Mark(26,(0,0))$
		$Mark(0,(0,0))$

0: Return: 0		Ok : $0 = 0$
1: Reset: [6] (2)	(0, 0)	
2: Fork: {3} (4, 8)	(0, 0)	
3: Parallel: (26, 25) <0>	–	Ok : $M = (3,1)$
4: Halt: <5> 1	(3, 1)	
5: Watchdog: 6 <3>	(3, 1)	
6: Test: [0] (7,5)	(3, 1)	$Mark(7,(3,1))$ $Mark(5,(3,1))$
8: Reset: [7] (9)	(3, 2)	
9: Fork: {10} (11, 17)	(3, 2)	
10: Parallel: (7, 24) <22>	(3, 2)	3 est père de 10 $Mark(7,(3,2))$ $Mark(24,(3,2))$ $Mark(22,(3,2))$
11: Halt: <12> 2	(10, 1)	$Mark(12,(10,1))$
17: Halt: <18> 3	(10, 2)	$Mark(18,(10,2))$
25: Emit: [4] (26)	(0, 0)	
26: Emit: [3] (0)	(0, 0)	Ok : $M = (0,0)$

0: Return: 0		
1: Reset: [6] (2)	(0, 0)	
2: Fork: {3} (4, 8)	(0, 0)	
3: Parallel: (26, 25) <0>	–	
4: Halt: <5> 1	(3, 1)	
5: Watchdog: 6 <3>	(3, 1)	Ok : $M = (3,1)$
6: Test: [0] (7,5)	(3, 1)	
7: Exit: {3} 0	(3, 1)	Ok : $M = (3,1)$, $3 = 3$ Ok : $M = (3,2)$, $3 = 3$
8: Reset: [7] (9)	(3, 2)	
9: Fork: {10} (11, 17)	(3, 2)	
10: Parallel: (7, 24) <22>	(3, 2)	
11: Halt: <12> 2	(10, 1)	
12: Watchdog: 13 <10>	(10, 1)	$Mark(13,(10,1))$ $Mark(10,(10,1))$
17: Halt: <18> 3	(10, 2)	
18: Watchdog: 19 <10>	(10, 2)	$Mark(19,(10,2))$ $Mark(10,(10,2))$
22: Watchdog: 23 <3>	(3, 2)	$Mark(23,(3,2))$ $Mark(3,(3,2))$
24: Emit: [5] (7)	(3, 2)	$Mark(7,(3,2))$
25: Emit: [4] (26)	(0, 0)	
26: Emit: [3] (0)	(0, 0)	

0: Return: 0		
1: Reset: [6] (2)	(0, 0)	
2: Fork: {3} (4, 8)	(0, 0)	
3: Parallel: (26, 25) <0>	–	Ok : $M = (3,2), 3 = 3$
4: Halt: <5> 1	(3, 1)	
5: Watchdog: 6 <3>	(3, 1)	
6: Test: [0] (7,5)	(3, 1)	
7: Exit: {3} 0	(3, 1)	Ok : $M = (3,2), 3 = 3$
8: Reset: [7] (9)	(3, 2)	
9: Fork: {10} (11, 17)	(3, 2)	
10: Parallel: (7, 24) <22>	(3, 2)	Ok : $M = (10,1), 10 = 10$ Ok : $M = (10,2), 10 = 10$
11: Halt: <12> 2	(10, 1)	
12: Watchdog: 13 <10>	(10, 1)	
13: Test: [0] (14,12)	(10, 1)	$Mark(14,(10,1))$ $Mark(12,(10,1))$
17: Halt: <18> 3	(10, 2)	
18: Watchdog: 19 <10>	(10, 2)	
19: Test: [1] (20,18)	(10, 2)	$Mark(20,(10,2))$ $Mark(18,(10,2))$
22: Watchdog: 23 <3>	(3, 2)	
23: Test: [2] (7,22)	(3, 2)	$Mark(7,(3,2))$ $Mark(22,(3,2))$
24: Emit: [5] (7)	(3, 2)	
25: Emit: [4] (26)	(0, 0)	
26: Emit: [3] (0)	(0, 0)	

0: Return: 0		
1: Reset: [6] (2)	(0, 0)	
2: Fork: {3} (4, 8)	(0, 0)	
3: Parallel: (26, 25) <0>	–	
4: Halt: <5> 1	(3, 1)	
5: Watchdog: 6 <3>	(3, 1)	
6: Test: [0] (7,5)	(3, 1)	
7: Exit: {3} 0	(3, 1)	Ok : $M = (3,2), 3 = 3$
8: Reset: [7] (9)	(3, 2)	
9: Fork: {10} (11, 17)	(3, 2)	
10: Parallel: (7, 24) <22>	(3, 2)	
11: Halt: <12> 2	(10, 1)	
12: Watchdog: 13 <10>	(10, 1)	Ok : $M = (10,1)$
13: Test: [0] (14,12)	(10, 1)	
14: Emit: [7] (15)	(10, 1)	$Mark(15,(10,1))$
17: Halt: <18> 3	(10, 2)	
18: Watchdog: 19 <10>	(10, 2)	Ok : $M = (10,2)$
19: Test: [1] (20,18)	(10, 2)	
20: Emit: [6] (21)	(10, 2)	$Mark(21,(10,2))$
22: Watchdog: 23 <3>	(3, 2)	Ok : $M = (3,2)$
23: Test: [2] (7,22)	(3, 2)	
24: Emit: [5] (7)	(3, 2)	
25: Emit: [4] (26)	(0, 0)	
26: Emit: [3] (0)	(0, 0)	

0: Return: 0		
1: Reset: [6] (2)	(0, 0)	
2: Fork: {3} (4, 8)	(0, 0)	
3: Parallel: (26, 25) <0>	–	
4: Halt: <5> 1	(3, 1)	
5: Watchdog: 6 <3>	(3, 1)	
6: Test: [0] (7,5)	(3, 1)	
7: Exit: {3} 0	(3, 1)	
8: Reset: [7] (9)	(3, 2)	
9: Fork: {10} (11, 17)	(3, 2)	
10: Parallel: (7, 24) <22>	(3, 2)	
11: Halt: <12> 2	(10, 1)	
12: Watchdog: 13 <10>	(10, 1)	
13: Test: [0] (14,12)	(10, 1)	
14: Emit: [7] (15)	(10, 1)	
15: Exit: {10} 2	(10, 1)	Ok : 10 = 10
17: Halt: <18> 3	(10, 2)	
18: Watchdog: 19 <10>	(10, 2)	
19: Test: [1] (20,18)	(10, 2)	
20: Emit: [6] (21)	(10, 2)	
21: Exit: {10} 3	(10, 2)	Ok : 10 = 10
22: Watchdog: 23 <3>	(3, 2)	
23: Test: [2] (7,22)	(3, 2)	
24: Emit: [5] (7)	(3, 2)	
25: Emit: [4] (26)	(0, 0)	
26: Emit: [3] (0)	(0, 0)	

Le code IC est correct. Notons que l'instruction 16 n'apparaît pas dans ce dernier tableau : elle n'est pas atteignable.

Exemple 2

Vérifions la correction des continuations du code IC suivant :

```
...
signals: 2
0: output: O1 $0 single: ;
1: output: O2 $0 single: ;
end:
...
statements: 4 startpoint: 1 ;
goloops: 0 ;
exitlevels: 0 ;
0: Return: 0 ;
1: Fork: {2} (3, 3) ;
2: Parallel: (0) <0> ;
3: Emit: [0] (4) ;
4: Exit: {2} 0 ;
end:
```

Instructions	M	Appels
`startpoint: 1` `0: Return: 0`		$Mark(1,(0,0))$
`0: Return: 0` `1: Fork: {2} (3, 3)`	$(0,0)$	$Mark(3,(2,1))$, $Mark(3,(2,2))$

`0: Return: 0`
`1: Fork: {2} (3, 3)`
`3: Emit: [0] (4)`
	$(0,0)$
	$(2,1)$
		Incorrect : $(2,2) \neq (2,1)$

La même instruction est exécutée par deux branches d'un même `Parallel` : le code est rejeté.

Exemple 3

Dans le code IC qui suit, l'instruction `Exit` 5 n'est pas liée au `Parallel` requis :

```
signals: 3
0: output: O1 $0 single: ;
1: output: O2 $0 single: ;
2: local: $0 single: ;
end:
...
statements: 13 startpoint: 1 ;
goloops: 0 ;
exitlevels: 2 ;
0: Return: 0 ;
1: Reset: [2] (2) ;
2: Fork: {3} (4, 6) ;
3: Parallel: (8, 0) <0> ;
```

```
4: Emit: [2] (5) ;
5: Exit: {9} 2 ;
6: Emit: [0] (7) ;
7: Exit: {3} 0 ;
8: Fork: {9} (10, 11) ;
9: Parallel: (0, 0) <0> ;
10: Emit: [0] (12) ;
11: Emit: [1] (12) ;
12: Exit: {9} 0 ;
end:
```

Instructions	M	Appels
`startpoint: 1` `0: Return: 0`		$Mark(1,(0,0))$
`0: Return: 0` `1: Reset: [2] (2)`	$(0,0)$	$Mark(2,(0,0))$

`0: Return: 0`
`1: Reset: [2] (2)`
`2: Fork: {3} (4, 6)`
	$(0,0)$
	$(0,0)$
		$Mark(4,(3,1))$ $Mark(6,(3,2))$

`0: Return: 0`
`1: Reset: [2] (2)`
`2: Fork: {3} (4, 6)`
`4: Emit: [2] (5)`
`6: Emit: [0] (7)`
	$(0,0)$
	$(0,0)$
	$(3,1)$
	$(3,2)$
		$Mark(5,(3,1))$
		$Mark(7,(3,2))$

`0: Return: 0`
`1: Reset: [2] (2)`
`2: Fork: {3} (4, 6)`
`4: Emit: [2] (5)`
`5: Exit: {9} 2`
`6: Emit: [0] (7)`
`7: Exit: {3} 0`
	$(0,0)$
	$(0,0)$
	$(3,1)$
	
	$(3,2)$
		Incorrect : $3 \neq 9$

Chapitre 5

Le format "graphe flot de données" : GC

5.1 Introduction

GC est une représentation hiérarchisée de style flot-de-données ou bloc-diagramme d'un programme synchrone, augmentée de contrôles explicites destinés notamment à accueillir le traitement de données des programmes traduits de IC. La partie déclarative d'un code GC a une sémantique synchrone, qui est une *spécification* au sens du modèle sémantique décrit en 5.7. Les notions d'horloge et de flot utilisées ci-après font référence à ce modèle. La sémantique complète de GC est présentée en annexe D.

5.1.1 Structure d'un code GC

Un code GC est constitué d'une liste de déclarations de trois sortes d'entités : des *blocs de données,* des *interfaces* et des *nœuds.*

5.1.2 Nœuds

L'entité de base d'un code GC s'appelle un *nœud.* Un nœud décrit dans le format GC comporte une interface, des données et un corps.

L'interface d'un nœud en décrit les propriétés visibles de l'extérieur.

Les données décrivent, sous la forme d'importations de blocs de données, les types, constantes, fonctions et procédures utilisés par le nœud.

Le corps d'un nœud est constitué de la donnée des composantes suivantes, décrites dans des tables spécifiques du nœud :

Définitions des valeurs des flots : Les expressions de définition expriment le calcul des valeurs des flots de sortie en fonction de flots d'entrées. Ce sont des *fonctions entrée/sortie déterministes,* elles obéissent au principe de substitution (leur membre de gauche peut partout être remplacé par leur expression de droite).

Les expressions de définition comportent des équations, des instanciations de nœuds, des appels de procédures et des activations de nœuds externes pouvant être des actions de module impératif.

Un ensemble d'expressions de définition peut être vu comme un graphe flot-de-données.

Il est possible d'ajouter à ce graphe des dépendances de contrôle explicites entre ces définitions.

Synchronisations : Les synchronisations spécifient des *contraintes* restreignant la sémantique du programme. Ce sont des relations et non pas des fonctions. Elles doivent être prises en compte par l'ensemble des outils, tant pour les preuves que pour la génération de code.

Assertions : Les assertions sont des propriétés énoncées sur les flots du nœud considéré (ce sont donc des relations et non des fonctions). Les assertions peuvent être utilisées comme *commentaires* et ignorées par les outils formels ou les générateurs de code : elles servent alors à décrire des hypothèses de fonctionnement de l'environnement, ces hypothèses pouvant alors éventuellement être vérifiées à la volée lors de l'exécution. Elles peuvent aussi être utilisées à des fins d'optimisation du code.

Dépendances : Les dépendances permettent de spécifier un ordre partiel d'exécution entre communications de flots à l'intérieur d'un même instant. Elles sont notamment nécessaires à la spécification des interfaces. Les dépendances peuvent varier d'un instant à l'autre, et ceci est codé par une étiquette qui est une *horloge* assignée à toute dépendance : la dépendance considérée est en vigueur lorsque les flots amont et aval de la dépendance considérée, et son horloge étiquette, sont tous présents. Certaines dépendances sont implicites (par exemple, les sorties de la plupart des opérateurs dépendent implicitement de leurs entrées), mais peuvent être explicitées en cours de compilation.

5.1.3 Interface d'un nœud, nœud externe

L'interface d'un nœud est le résumé requis pour l'utilisation de ce nœud en "boîte-noire" dans un contexte d'utilisation. L'interface d'un nœud est constituée :

d'une partie données qui décrit notamment, sous la forme d'importations de blocs de données, les types des flots d'entrée/sortie du nœud ;

de dépendances qui permettent de spécifier un ordre partiel de communication des flots d'interface à l'intérieur d'un même instant ;

d'assertions et de synchronisations qui sont des propriétés énoncées sur les flots d'interface du nœud considéré.

Un nœud externe est un nœud dont la description GC n'est pas accessible. Ce peut être en particulier la perspective GC de l'action d'un module impératif. Comme tout nœud, un nœud externe est assorti d'une interface qui en décrit les propriétés visibles de l'extérieur.

5.2 Organisation GC des données

5.2.1 Flots

Une table de flots est identifiée par le mot-clé **flows:**. On rappelle que la syntaxe générique des tables est décrite en 2.2.3.

♠ **Nom-table-flots** ::= • **flows:** •

Une entrée dans cette table contient la nature du flot (entrée **i:**, sortie **o:**, ou, implicitement, flot local), le nom du flot, son type, éventuellement sa valeur initiale et son horloge. L'horloge du flot est représentée par un index de flot qui doit être de type `$pure`.

♠ **OBJET-FLOT** ::= • **NATURE-DE-FLOT CANAL-DE-FLOT Horloge** •

♠ **NATURE-DE-FLOT** ::= • **i:** **Identificateur-*flot*** • [**o:**] **Identificateur-*flot*** •

♠ **CANAL-DE-FLOT** ::= • **Index-*type*** [**VALEUR-INITIALE-FLOT**] •

♠ **VALEUR-INITIALE-FLOT** ::= • **value:** **EXPRESSION** •

♠ **Horloge** ::= • **Index-*flot*** •

Associé à chaque nœud, il existe un flot prédéfini, au moins aussi rapide que tout autre flot visible dans le nœud, appelé `$base` et noté `$g0`, de type `$pure`. Ce flot est sa propre horloge.

Les flots d'un nœud sont :

- soit des flots d'interface et dans ce cas sont décrits dans la table des flots de cette interface ; ces flots doivent avoir un attribut de nature, **i:** ou **o:** ;
- soit des flots locaux, qui sont alors décrits dans la table des flots du nœud ; ces flots locaux n'ont pas d'attribut de nature.

5.2.2 Expressions spécifiques GC

Les expressions spécifiques au format GC sont les flots, qui complètent ainsi les expressions décrites en 2.2.5.

Un flot est simplement référencé par son index.

♠ **Terme-GC** ::= • **Index-*flot*** •

Dans un nœud (cf. 5.4, page 70) :

- les index de flots apparaissant dans une expression peuvent être soit des index locaux désignant des flots externes ou locaux du nœud, soit des index composés désignant des flots externes de nœuds importés ;
- les index de constantes apparaissant dans une expression peuvent être soit des index de constantes prédéfinies communes, soit des index composés désignant des constantes de blocs de données importés ;

- les index de fonctions apparaissant dans une expression peuvent être soit des index de fonctions prédéfinies communes, soit des index composés désignant des fonctions de blocs de données importés ou prédéfinis.

5.2.3 Types spécifiques GC

Types nouveaux

Pour chaque type μ différent du type **$pure**, et différent du type ν (pour tout type ν), un type **$win**($\mu$) peut être défini. La grammaire donnée en 2.4.1 est complétée par la règle suivante :

♠ **OBJET-TYPE** ::= • [**$win**] [( ] **Index-*type*** [ )] •

Les opérateurs associés sont présentés dans le bloc de données prédéfini GC.

Les entiers bornés sont du type entier et assujettis à une contrainte spécifiée par une assertion.

Forme de présentation

Pour cette présentation nous donnons une description du bloc de données puis de chacun des opérateurs. Pour chaque opérateur, nous donnons :

- les équations d'horloges qu'il induit implicitement (cf. 5.7, page 79) ;
- les dépendances qu'il induit implicitement **lorsqu'il est utilisé dans une définition** ; ces dépendances n'ont pas de contrepartie dans la sémantique de suites. Les dépendances sont des indications sur l'ordre d'évaluation au cours d'une réaction (donc *dans l'instant*) : on notera `X -h-> Y` le fait que, dans toute réaction où
  - l'horloge du flot `X` est présente,
  - l'horloge du flot `Y` est présente,
  - l'horloge `h` est présente (il s'agit implicitement de l'horloge **$base** lorsqu'elle n'est pas mentionnée),

  l'évaluation de `Y` ne peut précéder (dans l'instant!) l'évaluation de `X`. Pour les variantes de notations concernant les dépendances, se reporter à la sous-section 5.4.4 ;
- sa sémantique de suites (cf. 5.7, page 79).

Bloc de données prédéfini GC

Le bloc de données prédéfini GC contient notamment la définition des types génériques **$any** et **$win($g0)**. Les fonctions définies pour ces types sont surchargées : pour tout type μ, toute fonction définie pour le type **$any** est implicitement définie pour le type μ (chacune des références au type **$any** dans la définition de la fonction étant remplacée par une référence au même type μ). De la même façon, pour tout type μ pour lequel il existe un type **$win**($\mu$), toute fonction définie pour les types **$any** et **$win($g0)** est implicitement définie pour les types μ et **$win**($\mu$).

```
data: dc:0 GC
  types: 7
    $g0: $any;       -- aucune fonction associee
    $g1: $win($g0); -- aucune fonction associee
    $g2: $win($1);  -- $boolean   $g15 code $eq
    $g3: $win($2);  -- $integer   $g16 code $eq
    $g4: $win($3);  -- $string    $g17 code $eq
    $g5: $win($4);  -- $float     $g18 code $eq
    $g6: $win($5);  -- $double    $g19 code $eq
  end:
  functions: 20
    -- pure
    $g0: $tt ($1): $0;                    -- polychrone
    $g1: $clkadd ($0, $0): $0;            -- polychrone
    $g2: $clkdiff ($0, $0): $0;           -- polychrone
    $g3: $clkmult ($0, $0): $0;           -- polychrone
    $g4: $present ($0): $1;
    $g5: $clkeq ($0, $0): $1;             -- polychrone
    $g6: $clkimplies ($0, $0): $1;        -- polychrone
    $g7: $clkmutex ($0, $0): $1;          -- polychrone
    -- any
    $g8: $clock ($g0): $0;
    $g9: $default ($g0, $g0): $g0;        -- polychrone
   $g10: $when ($g0, $0): $g0;            -- polychrone
   $g11: $pre ($g0): $g0;
   $g12: $fby ($g0, $g0): $g0;
   $g13: $window ($g0, $2, $g1): $g1;
   $g14: $select ($g1, $2): $g0;
    --
   $g15: $eq ($g2, $g2): $1;
   $g16: $eq ($g3, $g3): $1;
   $g17: $eq ($g4, $g4): $1;
   $g18: $eq ($g5, $g5): $1;
   $g19: $eq ($g6, $g6): $1;
end:
enddata:
```

Le bloc de données prédéfini `GC` est importé implicitement par toute entité qui utilise un des objets qu'il définit.

Les références aux types `$win($1)`, `$win($2)`, `$win($3)`, `$win($4)`, `$win($5)` ont respectivement la forme "`$g2`", "`$g3`", "`$g4`", "`$g5`", "`$g6`". Une référence à une fonction d'index i dans le bloc de données `GC` a la forme "`$g`i".

Dans la suite nous pouvons être amenés à utiliser la forme littérale, plus lisible, des désignations d'objets prédéfinis (bien que cette forme soit syntaxiquement incorrecte).

5.2.4 Fonctions et procédures

Les paramètres des procédures et des fonctions sont des flots. Le résultat d'une fonction est également un flot. Si on appelle entrées les flots i: d'une procédure et les paramètres d'une fonction, sorties les flots o: d'une procédure et le résultat d'une fonction, les entrées et les sorties de chaque appel sont synchrones (ont même horloge) ; de plus toute entrée précède implicitement toute sortie à chaque appel.

Attention, ces relations implicites sur les horloges et les dépendances ne s'appliquent pas aux fonctions prédéfinies sur type pur (cf. 5.2.5, page 64) et aux fonctions prédéfinies génériques (cf. 5.2.6, page 66). Pour ces fonctions, les relations implicites sur les horloges et les dépendances sont spécifiées dans leur définition.

5.2.5 Fonctions prédéfinies sur type pur

Les flots de type **$pure** sont aussi appelés horloges.

La fonction $tt

- Horloge

$$h^{\$tt(X)} = \{h_n^X \mid v_n^X = true\}$$

- Dépendances
  La dépendance implicite suivante est engendrée : `X --> $tt(X)`

- Sémantique
  `$tt(X)` est le flot de type **$pure** présent si et seulement si le flot booléen `X` est présent et égal à *true*.

La fonction $clkadd

- Horloge

$$h^{\$clkadd(X,Y)} = h^X \cup h^Y$$

- Dépendances
  Les dépendances implicites suivantes sont engendrées : `(X,Y) --> $clkadd(X,Y)`

- Sémantique
  `$clkadd(X,Y)` est le flot de type **$pure** présent si et seulement si le flot de type **$pure** `X` ou le flot de type **$pure** `Y` est présent.

La fonction $clkdiff

- Horloge

$$h^{\$clkdiff(X,Y)} = h^X \ominus h^Y$$

- Dépendances
  Les dépendances implicites suivantes sont engendrées : `(X,Y) --> $clkdiff(X,Y)`

- Sémantique
  `$clkdiff(X,Y)` est le flot de type `$pure` présent si et seulement si le flot de type `$pure X` est présent et le flot de type `$pure Y` n'est pas présent.

La fonction `$clkmult`

- Horloge

$$h^{\texttt{\$clkmult(X,Y)}} = h^{\texttt{X}} \cap h^{\texttt{Y}}$$

- Dépendances
  Les dépendances implicites suivantes sont engendrées : `(X,Y) --> $clkmult(X,Y)`

- Sémantique
  `$clkmult(X,Y)` est le flot de type `$pure` présent si et seulement si le flot de type `$pure X` et le flot de type `$pure Y` sont présents.

La fonction `$present`

- Horloge
  `$present(X)` et `X` ont la même horloge.

$$h^{\texttt{\$present(X)}} = h^{\texttt{X}}$$

- Dépendances
  La dépendance implicite suivante est engendrée : `X --> $present(X)`

- Sémantique
  `$present(X)` est le flot booléen sur l'horloge `X`, dont la valeur est toujours égale à *true*.

$$\forall n \leq |h^{\texttt{X}}|,\ v_n^{\texttt{\$present(X)}} = true$$

La fonction `$clkeq`

- `$clkeq` n'est pas une fonction primitive :
  `$clkeq(X,Y)` est équivalente à l'expression suivante :
  `$default($present($clkmult(X,Y)),$not($present($clkadd(X,Y))))`

- Horloge

$$h^{\texttt{\$clkeq(X,Y)}} = h^{\texttt{X}} \cup h^{\texttt{Y}}$$

- Dépendances
  Les dépendances implicites engendrées se déduisent de l'expansion de la fonction.
  `(X,Y) --> $clkeq(X,Y)`

- Sémantique
  `$clkeq(X,Y)` est le flot booléen présent si et seulement si le flot de type `$pure X` ou le flot de type `$pure Y` est présent ; sa valeur est égale à *true* si et seulement si les flots `X` et `Y` sont tous deux présents :

$$\forall n \leq |h^{X} \cup h^{Y}|,$$

$$v_n^{\$clkeq(X,Y)} = \begin{cases} true & \text{si } h_n^{\$clkeq(X,Y)} \in h^{X} \cap h^{Y} \\ false & \text{sinon} \end{cases}$$

La fonction `$clkimplies`

- `$clkimplies` n'est pas une fonction primitive :
  `$clkimplies(X,Y)` est équivalente à l'expression suivante :

```
$default($present(Y),$not($present(X)))
```

- Horloge

$$h^{\$clkimplies(X,Y)} = h^{X} \cup h^{Y}$$

- Dépendances
  Les dépendances implicites engendrées se déduisent de l'expansion de la fonction.
  `(X,Y) --> $clkimplies(X,Y)`

- Sémantique
  `$clkimplies(X,Y)` est le flot booléen présent si et seulement si le flot de type `$pure X` ou le flot de type `$pure Y` est présent ; sa valeur est égale à *true* si et seulement si le flot `X` est au plus aussi souvent présent que le flot `Y` :

$$\forall n \leq |h^{X} \cup h^{Y}|,$$

$$v_n^{\$clkimplies(X,Y)} = \begin{cases} true & \text{si } h_n^{\$clkimplies(X,Y)} \in h^{Y} \\ false & \text{sinon} \end{cases}$$

La fonction `$clkmutex`

- `$clkmutex` n'est pas une fonction primitive :
  `$clkmutex(X,Y)` est équivalente à l'expression suivante :

```
$not($clkeq(X,Y))
```

  Horloges et dépendances s'en déduisent immédiatement.
  `$clkmutex(X,Y)` est le flot booléen présent si et seulement si le flot de type `$pure X` ou le flot de type `$pure Y` est présent ; sa valeur est égale à *true* si et seulement si au plus un des flots `X` ou `Y` est présent.

5.2.6 Fonctions prédéfinies génériques

La fonction `$clock`

- Horloge
  `$clock(X)` et `X` ont la même horloge.

$$h^{\texttt{\$clock(X)}} = h^{\texttt{X}}$$

- Dépendances
  `$clock(X)` n'engendre pas de dépendance implicite.

- Sémantique
  `$clock(X)` est le flot de type `$pure` présent si et seulement si le flot `X` est présent.

La fonction `$default`

- Horloge

$$h^{\texttt{\$default(X,Y)}} = h^{\texttt{X}} \cup h^{\texttt{Y}}$$

- Dépendances
  Soit h le flot pur d'horloge $h^{\texttt{Y}} \ominus h^{\texttt{X}}$, les dépendances implicites suivantes sont engendrées :
  `X -`$h^{\texttt{X}}$`-> $default(X,Y)`
  `Y -`h`-> $default(X,Y)`

- Sémantique
  `$default(X,Y)` est un flot de même type que les flots `X` et `Y`, présent si et seulement si le flot `X` ou le flot `Y` est présent ; sa valeur est égale à celle de `X` lorsque celui-ci est présent, et à celle de `Y` dans le cas contraire :

$$\forall n \leq |h^{\texttt{\$default(X,Y)}}|,$$
$$v_n^{\texttt{\$default(X,Y)}} = \begin{cases} v_{m_n}^{\texttt{X}} & \text{si } h_n^{\texttt{\$default(X,Y)}} = h_{m_n}^{\texttt{X}} \in h^{\texttt{X}} \\ v_{k_n}^{\texttt{Y}} & \text{si } h_n^{\texttt{\$default(X,Y)}} = h_{k_n}^{\texttt{Y}} \in h^{\texttt{Y}} \ominus h^{\texttt{X}} \end{cases}$$

La fonction `$when`

- Horloge

$$h^{\texttt{\$when(X,Y)}} = h^{\texttt{X}} \cap h^{\texttt{Y}}$$

- Dépendances
  La seule dépendance implicite engendrée est la suivante :
  `X -`$h^{\texttt{Y}}$`-> $when(X,Y)`

- Sémantique
  `$when(X,Y)` est un flot de même type que le flot `X`, présent si et seulement si le flot `X` et le flot `Y` (ce dernier ayant le type `$pure`) sont présents ; sa valeur est alors égale à celle de `X` :

$$\forall n \leq |h^{\texttt{\$when(X,Y)}}|,$$
$$v_n^{\texttt{\$when(X,Y)}} = v_{m_n}^{\texttt{X}} \text{ si } h_n^{\texttt{\$when(X,Y)}} = h_{m_n}^{\texttt{X}}$$

La fonction $pre

- Horloge

$$h^{\$pre(X)} = h^X$$

- Dépendances
  $pre(X) n'engendre pas de dépendance implicite.

- Sémantique
  $pre(X) est un flot égal au flot X, translaté d'une valeur d'indice 1 dans le passé :

$$\forall n \leq |h^{\$pre(X)}|,$$
$$v_n^{\$pre(X)} = \begin{cases} nil & \text{si } n = 1 \\ v_{n-1}^X & \text{si } n > 1 \end{cases}$$

où *nil* dénote la valeur indéfinie (non-initialisé).

La fonction $fby

- Horloge

$$h^{\$fby(X,Y)} = h^X = h^Y$$

- Dépendances
  Soient h le flot pur dont le seul instant de présence est le premier instant de présence des flots X et Y, et K le flot pur d'horloge $h^X \ominus h$, les dépendances implicites suivantes sont engendrées :
  X -h-> $fby(X,Y)
  Y -K-> $fby(X,Y)

- Sémantique
  $fby(X,Y) est un flot, de même type que celui de X et de Y, égal au flot Y, sauf au premier instant de leur horloge commune où sa valeur est celle de X :

$$\forall n \leq |h^{\$fby(X,Y)}|,$$
$$v_n^{\$fby(X,Y)} = \begin{cases} v_1^X & \text{si } n = 1 \\ v_n^Y & \text{si } n > 1 \end{cases}$$

La fonction $window

- Horloge

$$h^{\$window(X,N,W0)} = h^X$$

N est une constante entière strictement positive,
W0 est une constante vectorielle externe de longueur égale à N.

- Dépendances
  La dépendance implicite suivante est engendrée :
  `X --> $window(X,N,W0)`

- Sémantique
  L'expression "`$window(X,N,W0)`", où `X` est un flot de type `T`, est un flot `W`, de type `$win(T)` ; sa valeur est à chaque instant une suite de `N` éléments (on dira qu'il est de *taille* `N`), notés respectivement `W[1]`...`W[N]` ; à l'instant initial de son horloge, il a pour valeur `W0` ; à chaque instant distinct de cet instant initial, la propriété suivante est vérifiée :

  `W[N] = X` `W[k-1] = pre W[k]`

  Formellement, avec les notations précédentes, on a :

$$\forall n \leq |h^{\texttt{\$window(X,N,W0)}}|,$$

$$v_n^{\texttt{W}}[k] = \begin{cases} \text{si } k = \texttt{N} \text{ alors} & v_n^{\texttt{X}} \\ \text{si } 1 \leq k < \texttt{N} \text{ alors} & \begin{cases} \texttt{W0}[k] & \text{si } n = 1 \\ v_{n-1}^{\texttt{W}}[k+1] & \text{si } n > 1 \end{cases} \end{cases}$$

La fonction $select

- Horloge

$$h^{\texttt{\$select(WX,I)}} = h^{\texttt{WX}} = h^{\texttt{I}}$$

- Dépendances
  Considérant `N`, la *taille* de `WX`, et h l'horloge des instants de `I` où `I` est égal à `N`, les dépendances implicites engendrées sont les suivantes :
  `I --> $select(WX,I)`
  `WX -`h`-> $select(WX,I)`

- Sémantique
  L'expression "`$select(WX,I)`", où `WX` est un flot de type `$win(T)` et de *taille* `N`, et où `I` est un flot de valeurs entières strictement positives et au plus égales à `N`, est un flot de type `T`, ayant à chaque instant la valeur `WX[I]`.

  Formellement, avec les notations précédentes, on a :

$$\forall n \leq |h^{\texttt{\$select(WX,I)}}|,$$

$$v_n^{\texttt{\$select(WX,I)}} = v_n^{\texttt{WX}}[v_n^{\texttt{I}}]$$

5.3 Interfaces

L'interface d'un nœud spécifie une partie données (importation de blocs de données), ses flots d'entrée/sortie (flots externes) et les dépendances qui les concernent, et enfin les assertions et synchronisations portant sur ces flots externes en utilisant au besoin des flots intermédiaires qui apparaissent alors en fin de table des flots de l'interface, sans

attribut de nature. Comme les blocs de données, les interfaces sont déclarées séparément et peuvent être communes à plusieurs nœuds.

Une interface peut importer des blocs de données en en donnant les index externes dans sa table d'importations. Cette table ne peut pas comporter d'entités autres que des blocs de données.

♠ **INTERFACE** ::= • **interface:**
INFOS-INTERFACE
[TABLE-IMPORTATIONS]
PARTIE-DÉCLARATIVE
endinterface: •

♠ **INFOS-INTERFACE** ::= • **Format-gc Identificateur-***interface* [**flat:**] •

♠ **Format-gc** ::= • **gc:** **Numéro-version** •

♠ **PARTIE-DÉCLARATIVE** ::= • **[TABLE-FLOTS]**
[TABLE-ASSERTIONS]
[TABLE-SYNCHRONISATIONS]
[TABLE-DÉPENDANCES]
[TABLE-PRAGMAS] •

L'absence de l'une des tables optionnelles s'interprète comme l'absence de ses éléments constituants (pas de valeur par défaut).

Une interface sans table de flots ne contient aucune autre table. Ce peut être l'interface d'une action qui possède une horloge implicite `$base` comme seule horloge.

5.4 Nœuds

Un nœud est soit un nœud local, décrit dans le format GC, soit un nœud externe, c'est-à-dire dont la description GC n'est pas accessible.

♠ **NŒUD** ::= • **NŒUD-LOCAL** •
• **NŒUD-EXTERNE** •

5.4.1 Nœuds locaux

Une déclaration de nœud local consiste en :

- l'indication du format (**gc:**), avec son numéro de version ;
- le nom du nœud ;
- l'attribut optionnel **flat:** ;
- une **Propriété-de-processus** associée au nœud, spécifiée par les mots-clés **safe:**, **memorysafe:** ou **unsafe:** (cf. 2.4.5, page 26). En l'absence de cette spécification, le nœud est supposé *memorysafe* ;

- la table des références aux entités importées comprenant au moins, à l'index 0, l'index externe de l'interface qui décrit les propriétés externes du nœud (un nœud ne peut importer qu'une seule interface). En plus de son interface, la table des importations du nœud comporte les nœuds (locaux ou externes) utilisés par le nœud considéré, ainsi que d'éventuels blocs de données utilisés par le nœud (en particulier, les blocs de données importés par l'interface doivent être aussi importés par le nœud) ;
- la table des flots locaux du nœud ;
- une table des assertions portant sur des flots locaux ;
- une table des synchronisations internes ;
- une table des définitions des flots du nœud ;
- une table des dépendances ;
- une table des pragmas.

Sauf le nom, l'indication de format et la table d'importations, toutes les parties sont optionnelles.

♠ **NŒUD-LOCAL** ::= • **node:** **INFOS-NŒUD-LOCAL**
TABLE-IMPORTATIONS
PARTIE-DÉCLARATIVE
[TABLE-DÉFINITIONS]
endnode: •

♠ **INFOS-NŒUD-LOCAL** ::= • **Format-gc Identificateur-*nœud*** [**flat:**]
[Propriété-de-processus] •

5.4.2 Définitions

La table des définitions est identifiée par le mot-clé **definitions:**.

♠ **Nom-table-définitions** ::= • **definitions:** •

Nature des définitions

On trouve dans la table des définitions les expressions définissant les flots, soit directement sous la forme d'équations explicites, soit à travers les instanciations de nœud ; on y trouve également les appels aux procédures, les activations de nœud et les dépendances explicites entre définitions.

♠ **OBJET-DÉFINITION** ::= • **PARTIE-ÉQUATION** •
• **ACTIVATION** •
• **DÉPENDANCES-DÉFINITIONS** •

♠ **PARTIE-ÉQUATION** ::= • **ÉQUATION** •
• **INSTANCIATION** •
• **APPEL-PROCÉDURE** •

Équation explicite

Une équation définit le flot dont l'index apparaît en partie gauche par l'expression apparaissant en partie droite. Le flot et l'expression sont identifiés : ils ont même horloge, les prédécesseurs du flot sont ceux de l'expression, les horloges des dépendances sont conservées.

♠ **ÉQUATION** ::= • **define:** **Index-*flot*** **EXPRESSION** •

Les index de flots apparaissant en partie gauche d'une équation peuvent être soit des index locaux désignant des flots externes ou locaux du nœud, soit des index composés désignant des flots externes de nœuds importés.

Exemple : À l'index `10` de la table des définitions, définition de la valeur de `X`, flot local d'indice `41`, par l'addition entière de la constante 1 à la valeur de `Y`, flot local d'indice 4.

```
10: define: 41 $14(#1,4)
```

Exemplaire de nœud

Une instanciation de nœud définit les flots de sortie du nœud instancié ; les dépendances induites sont alors celles spécifiées dans l'interface du nœud. *Plusieurs occurrences syntaxiques d'instanciation d'un même nœud produisent des objets distincts.*

♠ **INSTANCIATION** ::=

• **set:** [**Horloge**] **Index-*nœud*** **(** **RENOMMAGES** **)** •

L'index du nœud instancié est un index local de la table des importations désignant un nœud importé.

L'horloge suivant le symbole **set:** est l'horloge de base de l'exemplaire produit. Elle doit être au moins égale à la borne supérieure des horloges de ses flots d'interface. En l'absence de cette horloge, celle-ci est implicitement égale à cette borne supérieure, mais n'est plus explicitement et directement accessible.

Exemple : À l'index `11` de la table des définitions, exemplaire du nœud dont la référence locale est l'index 7, sans renommage.

```
11: set:  7 ()
```

Appel de procédure

Un appel de procédure associe une horloge (dite *horloge d'activation)* et des flots à une occurrence d'utilisation syntaxique de procédure. Plusieurs occurrences syntaxiques d'appel d'une même procédure font référence au même objet, éventuellement à des horloges d'activation différentes et avec des flots différents : des effets de bord peuvent alors être produits si la procédure est *unsafe*.

♠ **APPEL-PROCÉDURE** ::=

• **call:** [**Horloge**] **Index-*procédure*** **(** [{ **EXPRESSION** **,** ... }] **)** •

L'index de la procédure appelée peut être soit un index de procédure prédéfinie commune, soit un index composé désignant une procédure d'un bloc de données importé.

L'occurrence d'horloge suivant le symbole `call:` est l'horloge d'activation de la procédure. Elle doit être au moins égale à la borne supérieure des horloges de ses flots d'interface. En l'absence de cette occurrence, la liste des expressions ne peut être vide, et l'horloge d'activation de la procédure est alors implicitement égale à la borne supérieure des flots d'interface.

Exemple : À l'index 12 de la table des définitions, appel, à l'horloge représentée par le flot pur d'index local 1414, d'une procédure dont la référence est l'index 1 dans la table des procédures du bloc de données d'index local 3, avec deux flots d'index locaux respectifs 41 et 5927 (environ).

```
12: call: 1414 3.1 (41, 5927)
```

Activation de nœud

Une activation de nœud est un appel à une catégorie particulière de procédures (actions) représentées par des nœuds externes (cf. 5.4.5, page 76) : elle associe donc une horloge d'activation et des flots à un tel nœud externe, qui représente la perspective GC d'une action d'un module impératif. *Contrairement à ce qui se passe pour une instanciation, plusieurs occurrences syntaxiques d'activation d'un même nœud font référence au même objet.* L'horloge de base du nœud externe est au moins égale à la borne supérieure de ses horloges d'activation.

♠ **ACTIVATION** ::=

• `do:` [**Horloge**] **Index-***nœud* `(` [{ **EXPRESSION** `,` ... }] `)` •

L'index du nœud activé est un index local de la table des importations, désignant un nœud importé.

Les expressions associent des flots de façon positionnelle aux flots externes du nœud activé, en respectant l'ordre sur ces flots tel que décrit en 5.3.

L'occurrence d'horloge suivant le symbole `do:` est l'horloge d'activation du nœud. Elle doit être au moins égale à la borne supérieure des horloges de ses flots d'interface. En l'absence de cette occurrence, l'horloge d'activation du nœud est implicitement égale à l'horloge de base du nœud qui contient cette activation.

Exemple : À l'index 13 de la table des définitions, activation, à l'horloge représentée par le flot pur d'index local 1414, d'une action sans paramètre désignée (cf. 5.4.5, page 76) dans le nœud externe dont la référence est l'index 3 dans la table des importations locale.

```
13: do: 1414 3 ()
```

Dépendances sur définition

Une dépendance sur définition ajoute (éventuellement), aux dépendances associées aux données, des précédences de contrôle entre les calculs exprimés dans cette table, que toute exécution devra respecter.

♠ **DÉPENDANCES-DÉFINITIONS** ::=

• **LISTE-INDEX-DÉFINITIONS LISTE-INDEX-DÉFINITIONS** [**Horloge**] •

♠ **LISTE-INDEX-DÉFINITIONS** ::= • **Index-*définition*** •
• [( ] { **Index-*définition*** [,] ... } [ )] •

Les index de définition sont des index locaux désignant des définitions de la table.

Exemple : À l'index 14 de la table des définitions, spécification, à l'horloge représentée par le flot pur d'index 732 dans l'entité d'index local 0, de la précédence de la définition de X ci-dessus sur l'activation d'index 13 ci-dessus.

```
14: 10 13 0.732
```

5.4.3 Renommages

Les objets des blocs de données (types, constantes, fonctions et procédures) et les flots externes déclarés dans son interface peuvent être renommés lors de l'instanciation d'un nœud. Par défaut ils ne le sont pas. Chaque renommage lors de l'instanciation d'un nœud P a la forme suivante :

$$\textit{catégorie: } NI_1/AI_1, \ldots, NI_n/AI_n$$

Tous les AI_j sont des index d'objets de la catégorie considérée, appartenant à l'interface ou à une table de données citée dans le nœud P ; tous les NI_j sont des index d'objets de la catégorie considérée, appartenant à l'interface ou à une table de données citée dans le nœud P ; chacun des objets AI_j est renommé en NI_j.

Les index AI_j sont des index relatifs au nœud instancié : les index de flots sont des index simples désignant directement des flots de l'interface du nœud instancié ; les index de types, constantes, fonctions et procédures sont des index composés désignant l'objet correspondant dans un bloc de données importé par le nœud instancié.

Les index NI_j sont des index relatifs au nœud courant (dans lequel se trouve l'instanciation) : les index de flots sont des index locaux désignant les flots locaux du nœud courant, ou des index composés désignant des flots de son interface ; les index de types, constantes, fonctions et procédures sont des index composés désignant l'objet correspondant dans un bloc de données importé par le nœud courant, ou des index d'objets prédéfinis.

Pour les flots, les NI_j peuvent être des expressions.

♠ **RENOMMAGES** ::= • [**RENOMMAGE-TYPES**]
[**RENOMMAGE-CONSTANTES**]
[**RENOMMAGE-FONCTIONS**]
[**RENOMMAGE-PROCÉDURES**]
RENOMMAGE-FLOTS •

♠ **RENOMMAGE-TYPES** ::=
• [**types:**] { **Index-*type*** [/] **Index-*type*** [,] ... } •

♠ **RENOMMAGE-CONSTANTES** ::=
• [**constants:**] { **Index-*constante*** [/] **Index-*constante*** [,] ... } •

♠ **RENOMMAGE-FONCTIONS** ::=
• [**functions:**] { **Index-*fonction*** [/] **Index-*fonction*** [,] ... } •

♠ **RENOMMAGE-PROCÉDURES** ::=

• **procedures:** { Index-*procédure* / Index-*procédure* , ... } •

♠ **RENOMMAGE-FLOTS** ::=

• **flows:** { **EXPRESSION** / Index-*flot* , ... } •

5.4.4 Dépendances de flots

Les dépendances implicites induites par les opérateurs ont été définies en même temps que ceux-ci.

Les dépendances implicites suivantes s'appliquent également :

- tout flot dépend implicitement de son horloge ;
- tout flot `X` dépendant d'un flot `Y` à une horloge h, dépend implicitement de cette horloge h.

Enfin, il est possible de définir des dépendances explicites entre flots, au moyen de la table des dépendances. En particulier, ces dépendances explicites sont utilisées dans les spécifications d'interface. La table des dépendances est identifiée par le mot-clé **dependences:**.

♠ **Nom-table-dépendances** ::= • **dependences:** •

♠ **OBJET-DÉPENDANCE** ::=

• **LISTE-INDEX-FLOTS LISTE-INDEX-FLOTS** [**Horloge**] •

♠ **LISTE-INDEX-FLOTS** ::= • Index-*flot* •
• ({ Index-*flot* , ... }) •

La sémantique de `X Y h` est que, à chaque instant pour lequel :

- le flot `X` est présent,
- le flot `Y` est présent,
- l'horloge `h` est présente (il s'agit implicitement de l'horloge `$base` lorsqu'elle n'est pas mentionnée),

l'évaluation (la connaissance de la valeur) de `X` doit précéder (dans l'instant !) l'évaluation de `Y`.

La notation

$$(X_1,\ldots,X_n)\ (Y_1,\ldots,Y_p)\ h$$

équivaut à

$$X_i\ Y_j\ h$$

pour tout couple (i,j).

5.4.5 Nœuds externes

Un nœud externe est un nœud dont la description GC n'est pas accessible ; l'index externe de son interface est fourni à l'indice 0 de la table d'importation.

Un tel nœud externe peut représenter la perspective GC d'une action d'un module impératif (**ic:**) ; l'index externe du module est dans ce cas fourni à l'indice 1 de la table d'importation. L'index de l'action IC suit immédiatement la table des importations.
Un nœud externe peut aussi représenter la perspective GC d'un objet complètement externe dont la description dans le format n'est pas accessible (**extern:**).

♠ **NŒUD-EXTERNE** ::= • **node:** **INFOS-NŒUD-EXTERNE-IMPÉRATIF**
TABLE-IMPORTATIONS
Index-*action*
[TABLE-PRAGMAS]
endnode: •
• **node:** **INFOS-NŒUD-EXTERNE**
TABLE-IMPORTATIONS
[TABLE-PRAGMAS]
endnode: •

♠ **INFOS-NŒUD-EXTERNE-IMPÉRATIF** ::=
• **Format-gc Identificateur-*nœud*** **ic:** **[Propriété-de-processus]** •

♠ **INFOS-NŒUD-EXTERNE** ::=
• **Format-gc Identificateur-*nœud*** **extern:** **[Propriété-de-processus]** •

5.5 Propriétés

5.5.1 Synchronisations

La table des synchronisations est identifiée par le mot-clé **synchronizations:**.

♠ **Nom-table-synchronisations** ::= • **synchronizations:** •

Les synchronisations spécifient des contraintes restreignant la sémantique du programme. Elles expriment des relations sur les flots, sous la forme :

- d'**EXPRESSIONS** à résultat booléen devant être toujours vrai lorsqu'il est présent ;
- d'**ÉQUATIONS**, d'**INSTANCIATIONS** ou d'**APPELS-PROCÉDURES**, qui spécifient des contraintes d'égalité d'horloges et de valeurs (**PARTIE-ÉQUATION**).

Une équation

```
define: E X
```

de la table des synchronisations équivaut dans cette table à l'expression

```
$clkeq($tt($eq(X,E)),$clock(X))
```

laquelle doit être toujours vraie.
La même relation est définie entre chaque flot de sortie X d'une instanciation ou d'un appel de procédure de la table des synchronisations et l'expression E qui le définit.

Les synchronisations ne génèrent pas de dépendances.

♠ **OBJET-SYNCHRONISATION** ::= • **OBJET-ASSERTION** •

5.5.2 Assertions

La table des assertions est identifiée par le mot-clé **assertions:**.

♠ **Nom-table-assertions** ::= • **assertions:** •

Comme les synchronisations, les assertions expriment des relations sur les flots, sous la forme :

- d'**EXPRESSIONS** à résultat booléen devant être toujours vrai lorsqu'il est présent ;
- d'**ÉQUATIONS**, d'**INSTANCIATIONS** ou d'**APPELS-PROCÉDURES**, qui spécifient des relations d'égalité d'horloges et de valeurs (cf. 5.5.1, page 76).

A la différence des synchronisations, les relations spécifiées par des assertions ne restreignent pas la sémantique du programme, mais décrivent des hypothèses de fonctionnement. Si $\Sigma_{\text{nœud}}$ et Σ_{assert} désignent respectivement les sémantiques de suites d'un nœud et de sa partie assertion, toute mise en œuvre satisfaisant à $\Sigma_{\text{nœud}}$ sous l'hypothèse que Σ_{assert} soit respecté est considérée comme correcte.

De même que les synchronisations, les assertions ne génèrent pas de dépendances.

♠ **OBJET-ASSERTION** ::= • **EXPRESSION** •
• **PARTIE-ÉQUATION** •

5.6 Exemple

Nous donnons ici un exemple de paquet GC.

```
package: 5 STOPWATCH

0: data: dc:0 STOPWATCH_DATA
   types: 1
   0: STOPWATCH_TIME_TYPE; -- 0 code $eq, 1 code $ne, 2 code $cond
   end:

   constants: 1
   0: INITIAL_STOPWATCH_TIME : 0;
   end:

   functions: 5
   0: $eq (0,0) : $1;
```

```
   1: $ne (0,0) : $1;
   2: $cond ($1,0,0) : 0;
   3: INCREMENT_STOPWATCH_TIME (0) : 0;
   4: IS_ZERO_MOD_10_MN (0) : $1;
   end:

enddata: -- STOPWATCH_DATA

1: interface: gc:0 TWO_STATES_INTF
   flows: 4
   0: i: init  $1 $g0;
   1: i: set   $1 $g0;
   2: i: reset $1 $g0;
   3: o: state $1 $g0;
   end:
   dependences: 3
   0: 0 3 $g0;
   1: 1 3 $g0;
   2: 2 3 $g0;
   end:
endinterface: -- TWO_STATES_INTF

2: node: gc:0 TWO_STATES
   import: 1
   0: 1;
   end:
   definitions: 1
   0: define:
       0.3 $g12(0.0,$3($5(0.1,$6($g11(0.3))),@$2,$3($5(0.2,$g11(0.3)),@$1,$g11(0.3))));
   end:
endnode: -- TWO_STATES

3: interface: gc:0 STOPWATCH_INTF
   import: 1
   0: 0;
   end:
   flows:7
   0: i: hs $1 $g0;
   1: i: start_stop $1 $g0;
   2: i: lap $1 $g0;
   3: o: time 0.0 $g0;
   4: o: run_state $1 $g0;
   5: o: lap_state $1 $g0;
   6: o: beep $2 $g0;
   end;
   assertions: 1
   0: $g7(0,1,2)
   end;
   dependences: 3
   0: 1 4;
   1: 2 5;
```

```
   2: (1,2,3) (3,6);
   end;
endinterface: -- STOPWATCH_INTF

4: node: gc:0 STOPWATCH
   import: 3
   0: 3;
   1: 0;
   2: 2;
   end:
   flows: 3
   0: reset $1 $g0;
   1: must_beep $1 $g0;
   2: internal_time 1.0 $g0;
   end:
   definitions: 7
   0: define: 0 $g12(@$1, $5(0.2,$g11($5($6(0.4),$6(0.5)))));
   1: set: 2 (flows: @$1/0,0.1/1,0.1/2,0.4/3);
   2: set: 2 (flows: @$1/0,$5(0.2,0.4)/1,0.2/2,0.5/3);
   3: define: 0.3 $g9($g10(2,$g8($6(0.5))),$g11(0.3));
   4: define: 2 1.2($g12(@$2,0), @1.0, 1.2($5(0.4,0.0), 1.3($g11(2)), $g11(2)));
   5: define: 1 $3(0.1,@$2,$3($5(0.0,0.4),1.4(2),@$1));
   6: define: 0.6 $9(1,#1,#0);
   end:
endnode: -- STOPWATCH

endpackage: -- STOPWATCH
```

5.7 Modèle sémantique du format GC

Le format GC permet de décrire des systèmes dynamiques. Un programme GC comporte essentiellement une partie déclarative (qui décrit un graphe de type schéma-bloc hiérarchisé), augmentée de contrôles explicites destinés notamment à traduire les séquences d'actions des programmes impératifs. Cette partie décrit le modèle sous-jacent de la partie déclarative. Celle-ci consiste en la *spécification* de l'ensemble des comportements légaux des *flots* concernés. Ces notions sont introduites et formalisées dans les deux sections suivantes. La sémantique complète de GC est décrite en annexe D.

5.7.1 Les Objets

Horloges, Flots, Traces : Une *horloge* est une suite strictement croissante, finie ou divergente, d'instants de $\mathbb{R}^+$, où $\mathbb{R}^+$ désigne l'ensemble des réels positifs ou nuls. Les opérations ensemblistes — union ($\cup$), intersection ($\cap$), différence ($\ominus$) — sont disponibles sur les horloges. On note $|h|$ le nombre d'éléments (fini ou dénombrable) de l'horloge h.

Un *flot* est une suite de valeurs synchronisée sur une horloge : tout flot X est donc un couple (h^X, v^X), où h^X est une horloge et v^X est une suite de valeurs du type de X, de même longueur que h^X. Intuitivement, X possède la valeur v_n^X à l'instant h_n^X. Une *trace* est un vecteur de flots $T = (X_1, \ldots, X_p)$.

contraintes d'horloge	signification
$h = k$	horloges égales
$h \# k$	horloges sans instant commun
$h \subset k$	les instants de h sont des instants de k

Tableau 5.1 : **Contraintes sur horloges**

Commentaire : Les valeurs particulières des instants d'une horloge sont sans signification, on en utilise seulement la propriété d'ordonnancement total. Par conséquent, si l'on considère un flot X pris isolément, seules sont pertinentes

- la longueur $|h^X|$ de son horloge,
- la suite $v_1^X, \ldots, v_n^X, \ldots$ de ses valeurs.

La notion d'horloge devient utile lorsque l'on considère plusieurs flots conjointement : on peut alors définir des *contraintes* sur leurs horloges, comme indiqué à la table 5.1. Ce degré de liberté est formalisé ci-après.

Changement de temps : On appelle *changement de temps* une bijection de $\mathbb{R}^+$ dans lui-même, préservant l'ordre et laissant 0 et $+\infty$ invariants. Par conséquent, les changements de temps transforment les horloges en horloges de même longueur.

5.7.2 Spécifications

Une *spécification* est simplement la donnée,

1. d'un ensemble fini A de noms de flots typés, ou "ports". Pour chaque $a \in A$, $\mathcal{D}_a$ désigne le domaine (ou type) des valeurs du flot X_a de nom a.
2. d'un sous-ensemble Σ de l'ensemble $\mathcal{T}_A$ de toutes les traces de la forme $(X_a)_{a \in A}$, où X_a est un flot de type D_a, ce sous-ensemble Σ satisfaisant à la propriété suivante :

$$\Sigma \textit{ est invariant par tout changement de temps} \tag{5.1}$$

Autrement dit, seul l'entrelacement global des horloges de l'ensemble des flots est significatif, les valeurs particulières prises par les instants ne le sont pas. Nous aurons à considérer deux opérateurs agissant sur les spécifications : la *restriction* et la *composition.*

La restriction : On se donne une spécification Σ portant sur l'ensemble de flots A, et on considère un sous-ensemble de ports $A' \subset A$. La restriction de Σ à A' est notée $\Sigma_{!!A'}$; elle est constituée de l'ensemble des traces $T' = (X_a)_{a \in A'}$ lorsque $T = (X_a)_{a \in A}$ parcourt Σ. On notera que $\Sigma_{!!A'}$ satisfait bien à la propriété (5.1).

La composition : Considérons deux spécifications Σ_1 et Σ_2, respectivement définies sur les ensembles de ports A_1 et A_2. Alors, $\Sigma_1|\Sigma_2$ désigne la spécification maximale[1] Σ définie sur l'ensemble de ports $A = A_1 \cup A_2$, satisfaisant les conditions suivantes :

$$\Sigma_{!!A_1} \subseteq \Sigma_1$$
$$\Sigma_{!!A_2} \subseteq \Sigma_2$$

Autrement dit, la composition contraint les flots de Σ_1 et Σ_2 de même nom à être égaux.

Exemple : Nous donnons une illustration de la notion de composition dans un cas où les spécifications Σ_1 et Σ_2 sont, chacune, explicitées par un petit programme élémentaire tel qu'on peut l'écrire en GC. Σ_1 est donnée par l'équation GC

```
Y := X when tt(X>0)
```

qui signifie que `Y` est présent et porte la même valeur que `X` lorsque la condition `X>0` est satisfaite, et est absent dans le cas contraire (l'opérateur GC "`tt(b)`" délivre l'horloge dont les instants de présence sont ceux où le flot booléen `b` est présent et porte la valeur *"vrai"*, tandis que "`X when h`" transmet les valeurs de `X` lorsque l'horloge `h` est présente). Cette instruction correspond à la spécification Σ_1 suivante, portant sur les flots `X,Y` :

$$h^Y = \{h_n^X \mid v_n^X > 0\}$$
$$\forall n \leq |h^Y|,\ v_n^Y = v_{m_n}^X,\ \text{où } m_n \text{ est l'indice tel que } h_{m_n}^X = h_n^Y$$

D'autre part, Σ_2 est donnée par l'équation

```
Z := Y + U
```

dont la signification est que les flots `U`, `Y` et `Z` sont présents simultanément (ils ont donc la même horloge), et que, lorsqu'ils sont présents, le troisième est la somme des deux premiers. La spécification $\Sigma_1|\Sigma_2$ résultant de la composition des deux spécifications Σ_1 et Σ_2, qui sera simplement représentée par le programme GC

```
Y := X when tt(X>0)
Z := Y + U
```

est donc donnée par les équations

$$h^Y = h^Z = h^U = \{h_n^X \mid v_n^X > 0\}$$
$$\forall n \leq |h^Y|, \begin{cases} v_n^Y = v_{m_n}^X \\ v_n^Z = v_n^Y + v_n^U \end{cases}$$

et peut donc être interprétée de manière intuitive comme suit :

> `Y`, `Z` et `U` ne sont présents qu'aux instants où `X` est présent et porteur d'une valeur positive, et à ces instants on a `Y = X` et `Z = Y + U`.

En particulier, l'effet de cette composition sur l'entrée `U` de ce programme peut être interprété comme un "blocage en lecture" de `U` dans l'attente de la prochaine occurrence positive de `X`, et ce sans perte d'occurrence présente de ce flot `U`.

[1] pour l'inclusion $\Sigma' \subseteq \Sigma$ définie sur les spécifications

Dans la présentation de GC, la sémantique des objets GC est donnée en termes de spécifications au sens ci-dessus. Une spécification est définie au moyen des informations suivantes :

1. **Liste et types des flots concernés** : la spécification considérée est alors un sous-ensemble de toutes les traces à valeur dans les domaines ainsi introduits.

2. **Équations d'horloges** : elles expriment les horloges associées aux divers flots, ainsi que les relations entre ces horloges.

3. **Équations de flots** : elles donnent les équations devant être satisfaites entre les valeurs présentes des flots considérés, avec mention des instants où ces équations sont en vigueur.

   Les équations de flots sont précédées chacune d'une expression d'horloge, qui spécifie les instants où cette équation est en vigueur.

   Par exemple, on écrit

   $$h_X = h_Y = h_Z \ , \ \forall n \leq |h_X| : \ v_n^Z = v_n^X + v_n^Y$$

   pour indiquer que `X`, `Y`, et `Z` ont même horloge et qu'aux instants de cette horloge, la valeur de `Z` est la somme de celles de `X` et de `Y`.

   De même, on écrit

   $$h_Z = h_X \cup h_Y \ , \ \forall n \leq |h_Z|, v_n^Z = \begin{cases} v_{m_n}^X, & \text{si } h_n^Z = h_{m_n}^X \in h^X \\ v_{k_n}^Y, & \text{si } h_n^Z = h_{k_n}^Y \in h^Y \ominus h^X \end{cases}$$

   pour indiquer que `Z` est présent lorsque `X` ou `Y` l'est, et que `Z=X` lorsque `X` est présent, et que `Z=Y` si `X` est absent et `Y` présent.

La composition de spécifications (opérateur |) étant évidemment associative et commutative, on peut obtenir la sémantique du corps d'un nœud GC, soit en suivant la méthode indiquée ci-dessus globalement pour le nœud considéré, soit en construisant d'abord la sémantique de chacune des équations d'horloges ou de flots, puis en composant ces sémantiques. La sémantique d'un nœud GC s'obtient alors en appliquant à la spécification correspondant au corps l'opérateur de restriction que nous avons défini sur les spécifications, de façon à ne laisser visibles que les ports d'interface du nœud.

Chapitre 6

Le format impératif séquentiel : OC

6.1 Introduction

Dans cette partie, nous présentons les deux tables particulières du format OC : la table des *dags* (*Directed Acyclic Graphs*) et celle des états. Ces deux tables apparaissent dans cet ordre. Mis à part ces deux tables, qui remplacent la table des instructions IC et représentent le résultat de la compilation de ces instructions en automate, les formats IC et OC sont identiques.

La première section présente la syntaxe hors-contexte d'un module OC. La deuxième section décrit ensuite le codage des automates en OC et présente différentes représentations plus ou moins compactes. La section suivante expose la grammaire complète du codage des *dags* utilisés par cette représentation, tandis que les deux dernières sections décrivent respectivement la syntaxe exacte de la table des *dags* et de la table des états.

6.2 Structure du format OC

Le format OC a une structure semblable à celle du format IC et sa syntaxe hors-contexte est la suivante :

♠ **MODULE-OC** ::= • **module:** **INFOS-MODULE-OC**
PARTIE-COMMUNE-IMPÉRATIVE
[TABLE-DES-DAGS]
TABLE-DES-ÉTATS
endmodule: •

♠ **INFOS-MODULE-OC** ::=
• **Format-oc** **Identificateur-***module* **[flat:]** **[Propriété-de-processus]** •

♠ **Format-oc** ::= • **oc:** **Numéro-version** •

On rappelle que la **PARTIE-COMMUNE-IMPÉRATIVE** est définie dans le chapitre décrivant le format IC(cf. 4.2, page 39).

6.3 Codage des automates en OC

6.3.1 Premier codage

Un automate est constitué d'un ensemble d'états et d'un ensemble de transitions entre ces états. Chaque transition est elle-même constituée d'une suite d'actions à effectuer et d'une instruction de changement d'état. Toutes les transitions partant d'un même état sont codées par un arbre de décision dont les nœuds sont les actions à effectuer (représentées par leur index dans la table des actions) et les feuilles sont les instructions de changement d'état. Les nœuds sont de deux types, distingués par le nombre de fils :

- les nœuds ayant un seul fils sont des actions pures (cf. 3.3, page 35) ;
- les nœuds ayant deux fils sont des actions de test (cf. 3.3, page 35), chaque fils représentant la suite d'une transition selon que le test est vrai ou faux.

Le codage de ces arbres de décision en OC se fait de la manière suivante :

- les actions sont codées par leur index dans la table des actions ;
- une instruction de changement d'état est codé par [<] **Index-*état*** [>] ;

Les nœuds n'ayant qu'un seul fils sont simplement écrits les uns à la suite des autres tandis que, pour les nœuds de test, on écrit d'abord l'index de l'action de test puis, entre parenthèses, la branche à exécuter si le test est vrai et, toujours entre parenthèses, la branche à exécuter si le test est faux.

A titre d'exemple, la figure 6.1 montre un extrait d'un programme ESTEREL et la

```
present S1 then
   present S2 then
      emit O1
   else
      await I1
   end present
else
   emit O2
end present;
emit O3;
await I2
```

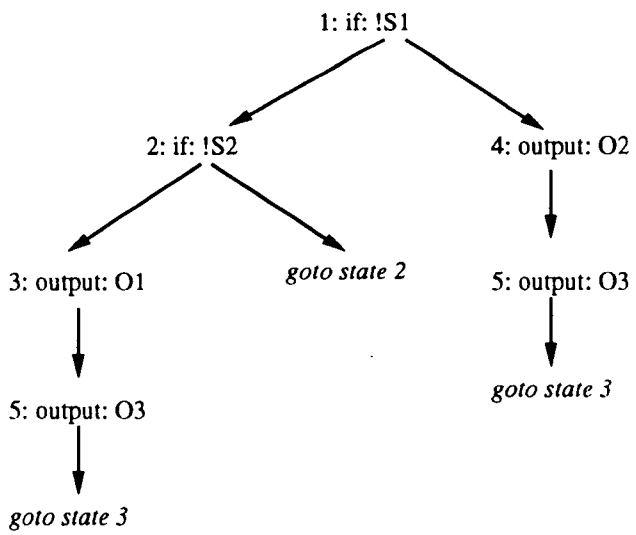


Figure 6.1 : Un extrait de programme ESTEREL et l'arbre de décision associé

partie correspondante de l'arbre de décision généré par ce programme. Afin de faciliter la lecture, les actions de l'arbre de décision comportent à la fois leur index et le libellé complet. De plus, les références aux signaux sont faites à l'aide des noms des signaux et non pas de leur index dans la table des signaux. Cet arbre de décision est codé en OC à l'aide des index d'actions de la manière suivante :

```
1 (2 (3 5 <3>) (<2>)) (4 5 <3>)
```

Les arbres de décision sont regroupés dans la table des états où chaque état est simplement représenté par l'arbre de décision correspondant (cf. 6.6, page 88).

6.3.2 Transformation des arbres de décision en *dags*

Une première modification de cette représentation permet d'obtenir un codage plus compact. Lorsque la fin des deux branches d'un nœud de test d'un arbre sont identiques, on peut les partager en transformant l'arbre de décision en un graphe acyclique orienté (aussi appelé *dag*, pour *Directed Acyclic Graph*). Les instructions de changement d'état arrêtant l'exécution, on peut même partager la fin de branches n'appartenant pas directement au même test (comme, par exemple, la partie "`5: output: O3; goto state 3`" dans l'exemple précédent), mais on interdit que les deux branches d'un même test partagent chacune une continuation différente avec d'autres branches, afin de conserver des *dags* simples.

Par exemple, l'arbre de décision de l'extrait de programme ESTEREL précédent peut être transformé dans le *dag* de la figure 6.2.

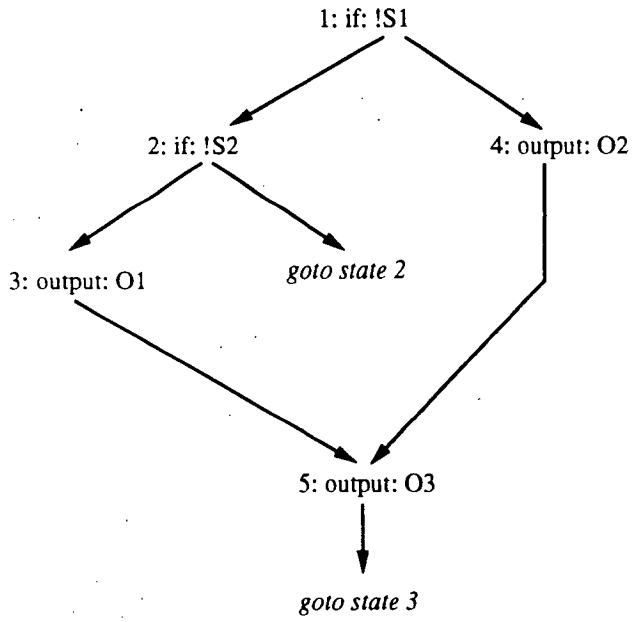


Figure 6.2 : *dag* associé à l'extrait de programme ESTEREL

Grâce à leur structure particulière, les *dags* obtenus par transformation des arbres de décision peuvent encore être représentés par des arbres. En effet, chaque partie partagée par deux ou plusieurs branches d'un arbre de décision peut être associée au test dont dépendent les branches qui partagent cette partie (dans l'exemple ci-dessus, la continuation "`5: output: O3; goto state 3`" peut être associée au test "`1: if: !S1`"). De tels tests sont alors représentés non plus par des nœuds ayant deux fils, mais par des nœuds ayant trois fils : le troisième fils est alors la continuation commune de toutes les branches qui ne sont pas terminées par des instructions de changement d'état. Pour coder ces nœuds à trois fils en OC, on écrit d'abord l'index de l'action de test puis, entre parenthèses, la branche à exécuter si le test est vrai puis, toujours entre parenthèses, la branche

à exécuter si le test est faux et, enfin, directement à la suite, l'éventuelle continuation. Ainsi, le *dag* de l'exemple précédent est codé en OC de la manière suivante :

```
1 (2 (3) (<2>)) (4) 5 <3>
```

Bien sûr, lorsqu'il y a plusieurs test imbriqués dans un *dag*, une même transition peut utiliser les continuations successives de plusieurs tests.

6.3.3 Partage des sous-*dags*

Étant donné un nœud d'un *dag*, on définit récursivement le "sous-*dag*" ayant ce nœud pour *racine* de la façon suivante : c'est l'ensemble composé de la racine et des nœuds dont tous les pères sont dans le sous-*dags*. Par exemple, la partie "`2: if: !S2, 3: output:O1, goto state 2`" est un sous-*dag* (dont la racine est "`2: if: !S2`") du *dag* de l'exemple précédent, mais pas la partie "`1: if: !S1, 4: output: O2, 5: output: O3`". Un sous-*dag* sera dit "fermé" si et seulement si toutes ses feuilles sont des instructions de changement d'état. Dans le cas contraire, le sous-*dag* sera dit "ouvert". Remarquons au passage que tous les *dags* associés aux états de l'automate sont fermés.

Une seconde modification de la représentation des *dags* permet d'obtenir un codage encore plus compact. Lorsqu'un sous-*dag* (ouvert ou fermé) apparaît plusieurs fois dans un ou plusieurs *dags*, on peut le partager en remplaçant toutes ses occurrences par une référence à une instance unique de ce sous-dag. Cette instance unique est placée dans la table des *dags* et la référence est appelée une "instruction de changement de *dag*". On distingue deux types d'instructions de changement de *dag*, selon que le *dag* référencé est ouvert ou fermé (ce qui permet de vérifier localement la fermeture d'un *dag*). Les instructions indiquant un changement vers un *dag* ouvert sont codées en OC par `{` **Index**-*dag* `}`, tandis que celles indiquant un changement vers un *dag* fermé sont codées par `[` **Index**-*dag* `]`.

Ainsi, le *dag* de l'exemple précédent peut être représenté par les deux *dags* de la figure 6.3. Ce *dag* est codé en OC de la manière suivante :

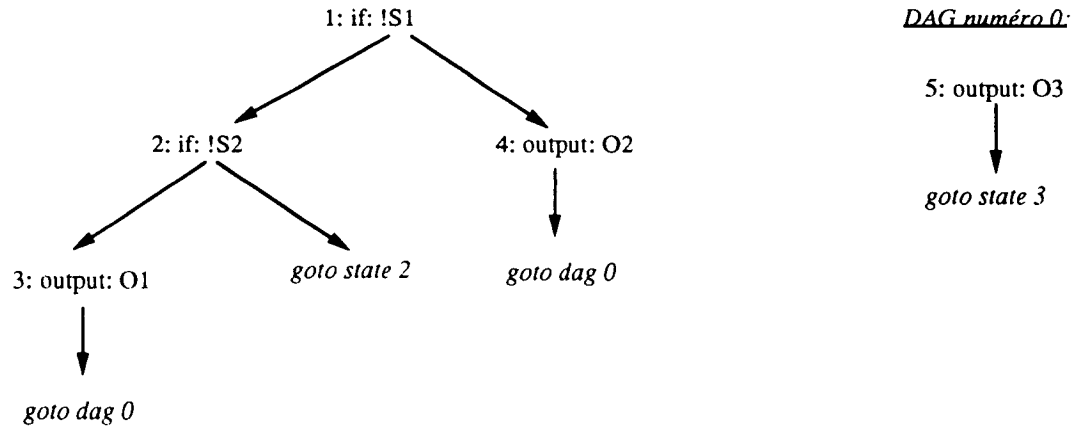


Figure 6.3 : *dags* associés à l'extrait de programme ESTEREL

```
1 (2 (3 [0]) (<2>)) (4 [0])
```

où le *dag* (fermé) d'index 0 est le suivant :

```
5 <3>
```

6.4 Table des *dags*

La table des *dags* est composée :

- du mot-clé **dags:** identifiant la table, suivi du nombre de *dags* dans la table ;
- de l'information supplémentaire indiquant le nombre total d'appels d'actions faits dans les *dags* de la table (mot-clé **calls:** suivi d'un entier et du caractère ;) ;
- de la liste des *dags* de la table, chaque *dag* étant précédé de son numéro et du caractère : et suivi du caractère ; ;
- du mot-clé **end:**.

♠ **Nom-table-dags** ::= • **dags:** •

♠ **INFOS-TABLE-DAGS** ::= • **calls:** **Cst-entière** ; •

6.5 Grammaire des *dags*

Pour résumer, voici la syntaxe complète des *dags* que l'on peut trouver dans la table des états ou la table des *dags* :

♠ **OBJET-DAG** ::= • **DAG-FERMÉ** •
• **DAG-OUVERT** •

♠ **DAG-FERMÉ** ::= • **DAG-OUVERT** **FERMETURE-DE-DAG** •

♠ **DAG-OUVERT** ::= • [**LISTE-DE-DAGS**] [{ **Index-***action-pure* ... }] •

♠ **LISTE-DE-DAGS** ::= • { **ACTIONS-ET-TEST-OUVERT** ... } •

♠ **ACTIONS-ET-TEST-OUVERT** ::= • { **Index-***action-pure* ... } **TEST-OUVERT** •

♠ **FERMETURE-DE-DAG** ::= • < **Index-***état* > •
• [**Index-***dag*] •
• **TEST-FERMÉ** •

♠ **TEST-FERMÉ** ::=
• **Index-***action-de-test* (**DAG-FERMÉ**) (**DAG-FERMÉ**) •

♠ **TEST-OUVERT** ::=

- • [{] **Index-***dag* [}] •
- • **Index-***action-de-test* [( ] **DAG-OUVERT** [ )] [( ] **DAG-FERMÉ** [ )] •
- • **Index-***action-de-test* [( ] **DAG-FERMÉ** [ )] [( ] **DAG-OUVERT** [ )] •
- • **Index-***action-de-test* [( ] **DAG-OUVERT** [ )] [( ] **DAG-OUVERT** [ )] •

6.6 Table des états

La table des états est composée :

- du mot-clé [**states:**] identifiant la table, suivi du nombre d'états dans la table ;
- des informations supplémentaires indiquant :
  - l'index de l'état de départ (mot-clé [**startpoint:**] suivi d'un index et du caractère [;]) ;
  - éventuellement, l'index de l'état de puits (mot-clé [**sink:**] suivi d'un index et du caractère [;]) ;
  - le nombre total d'appels d'actions faits dans les *dags* de la table (mot-clé [**calls:**] suivi d'un entier et du caractère [;]) ;
- de la liste des états de la table, chaque état étant composé de son numéro suivi du caractère [:], du *dag* codant les transitions partant de cet état et du caractère [;] ;
- du mot-clé [**end:**].

♠ **Nom-table-états** ::= • [**states:**] •

♠ **INFOS-TABLE-ÉTATS** ::= • [**startpoint:**] **Index-***état-de-départ* [;]
[[**sink:**] **Index-***état-puits* [;]]
[**calls:**] **Cst-entière** [;] •

♠ **OBJET-ÉTAT** ::= • **DAG-FERMÉ** •

Annexe A

Liste de pragmas

Le terminal **Valeur-pragma** dans le format est en fait muni d'une syntaxe, qui dépend du nom de pragma ; cette syntaxe peut être définie à l'aide des non terminaux et terminaux de la grammaire du format. Pour sa définition le terminal **Valeur-pragma** est ici considéré comme un non terminal **VALEUR-PRAGMA**.

A.1 Pragmas communs

A.1.1 Pragma-entité "main"

Le pragma **main:** est une directive de compilation qui peut être associée à une entité : celle-ci est alors une entité principale du programme.

♠ **Nom-pragma** ::= • **main:** •

♠ **VALEUR-PRAGMA** ::= • •

A.1.2 Pragma-commentaire "comment"

Le pragma-commentaire est notamment utilisé pour transporter des commentaires du code source à travers les codes intermédiaires. Sa syntaxe est la suivante :

♠ **Nom-pragma** ::= • **comment:** •

♠ **VALEUR-PRAGMA** ::= • **Cst-chaîne** •

La constante chaîne de caractères est le commentaire.

Ce pragma peut se trouver en un nombre quelconque d'exemplaires dans n'importe quel endroit d'un paquet. Il n'a pas de valeur par défaut ; son nom ne peut donc pas figurer dans la liste des pragmas calculés d'un module.

A.2 Pragmas apparaissant dans les modules IC et OC

A.2.1 Pragma-fichier "file"

Le pragma-fichier ne peut apparaître que dans une table d'instances et est utilisé pour désigner le fichier-source dans lequel se trouve une instance donnée. Sa syntaxe est la suivante :

♠ **Nom-pragma** ::= • **file:** •

♠ **VALEUR-PRAGMA** ::= • **Cst-chaîne** •

La constante chaîne de caractères est le nom du fichier.

Ce pragma ne peut être associé qu'à des entrées dans une table d'instances, en au plus un exemplaire pour chaque entrée. Il n'a pas de valeur par défaut ; son nom ne peut donc pas figurer dans la liste des pragmas calculés d'un module.

A.2.2 Pragma-répertoire "dir"

Le pragma-répertoire ne peut apparaître que dans une table d'instances et est utilisé pour désigner le répertoire dans lequel se trouve le fichier-source dans lequel se trouve une instance donnée. Sa syntaxe est la suivante :

♠ **Nom-pragma** ::= • **dir:** •

♠ **VALEUR-PRAGMA** ::= • **Cst-chaîne** •

La constante chaîne de caractères est le nom du répertoire.

Ce pragma ne peut être associé qu'à des entrées dans une table d'instances, en au plus un exemplaire pour chaque entrée. Il n'a pas de valeur par défaut ; son nom ne peut donc pas figurer dans la liste des pragmas calculés d'un module.

A.2.3 Pragma-source "lc"

Le pragma-source "lc" est utilisé pour pointer un endroit dans un texte-source ("lc" signifie "ligne-caractère", ou "ligne-colonne" si tous les caractères ont la même largeur). Sa syntaxe est la suivante :

♠ **Nom-pragma** ::= • **lc:** •

♠ **VALEUR-PRAGMA** ::= • **Cst-entière Cst-entière Index** •

Les deux constantes entières et l'index représentent respectivement :

- le numéro de la ligne dans le texte ;
- le numéro du caractère (ou de la colonne) dans la ligne ;

- l'index de l'instance dans la table des instances.

Les lignes et les colonnes sont numérotées à partir de 1.

Ce pragma peut se trouver en un nombre quelconque d'exemplaires dans n'importe quelle table d'un module. Il désigne l'endroit du texte-source qui a produit ou contribué à produire l'objet auquel il est associé. Il n'a pas de valeur par défaut ; son nom ne peut donc pas figurer dans la liste des pragmas calculés d'un module.

A.2.4 Pragma-source "instance"

Ce pragma est identique au pragma-source "`lc`" mais il ne peut être associé qu'à une entrée de la table des instances. Il indique l'endroit où est réalisée l'instance, c'est-à-dire l'endroit dans le module-père où l'instance à laquelle le pragma est associé est utilisée. La syntaxe de ce pragma est la suivante :

♠ **Nom-pragma** ::= • **instance:** •

♠ **VALEUR-PRAGMA** ::= • **Cst-entière Cst-entière Index** •

A.2.5 Pragma "name"

Le pragma-nom est utilisé pour associer un nom à un objet. Sa syntaxe est la suivante :

♠ **Nom-pragma** ::= • **name:** •

♠ **VALEUR-PRAGMA** ::= • **Identificateur** •

Ce pragma peut être associé à n'importe quel objet d'un paquet mais tout objet ne peut avoir au plus qu'un seul nom. Si plusieurs noms doivent être associés à un objet, il faut utiliser le pragma "`alias`" décrit ci-dessous.

Ce pragma n'a pas de valeur par défaut ; son nom ne peut donc pas figurer dans la liste des pragmas calculés d'un module.

A.2.6 Pragma "alias"

Le pragma-alias est utilisé pour associer une liste de noms à un objet. Sa syntaxe est la suivante :

♠ **Nom-pragma** ::= • **alias:** •

♠ **VALEUR-PRAGMA** ::= • **Identificateur** [{ **Identificateur** ... }] •

Ce pragma peut être associé à n'importe quel objet d'un paquet. Il n'a pas de valeur par défaut ; son nom ne peut donc pas figurer dans la liste des pragmas calculés d'un module. Il faut noter qu'aucun nom de la liste n'est distingué. Si un nom particulier doit être associé à un objet, il faut utiliser le pragma "`name`" décrit ci-dessus.

Attention : ce pragma peut-être très long (plusieurs milliers de caractères !) dans le cas de gros exemples. Prévoir les tailles des buffers des analyseurs lexicaux en conséquence...

A.2.7 Pragma "previous"

Le pragma "previous" est utilisé pour indiquer le "père" d'un signal dans l'arborescence des déclarations de signaux. Sa syntaxe est la suivante :

- **Nom-pragma** ::= • **previous:** •
- **VALEUR-PRAGMA** ::= • **Index** •

La valeur "-" signifie que le signal est la racine de l'arborescence des déclarations de signaux. Sinon, l'index désigne le signal-père dans la table des signaux. Pour le moment, ce pragma n'a de sens que dans la table des signaux lorsqu'il est associé à une entrée de cette table. Il n'a pas de valeur par défaut ; son nom ne peut donc pas figurer dans la liste des pragmas calculés d'un module.

A.2.8 Pragma "sigbool"

Le pragma "sigbool", associé à une entrée dans la table des variables, est utilisé pour indiquer que cette variable est le booléen de présence d'un signal. Sa syntaxe est la suivante:

- **Nom-pragma** ::= • **sigbool:** •
- **VALEUR-PRAGMA** ::= • **Index** •

L'index désigne le signal auquel la variable (nécessairement de type booléen $1) est associée dans la table des signaux. Ce pragma n'a pas de valeur par défaut ; son nom ne peut donc pas figurer dans la liste des pragmas calculés d'un module.

A.2.9 Pragma "sigval"

Le pragma "sigval", associé à une entrée dans la table des variables, est utilisé pour indiquer que cette variable contient la valeur courante d'un signal. Sa syntaxe est la suivante:

- **Nom-pragma** ::= • **sigval:** •
- **VALEUR-PRAGMA** ::= • **Index** •

L'index désigne le signal auquel la variable (du même type que le signal) est associée dans la table des signaux. Ce pragma n'a pas de valeur par défaut ; son nom ne peut donc pas figurer dans la liste des pragmas calculés d'un module.

A.2.10 Pragma "count"

Le pragma "count", associé à une entrée dans la table des variables, est utilisé pour indiquer que cette variable a été créée par le compilateur et implémente un compteur. Sa syntaxe est la suivante :

♠ **Nom-pragma** ::= • **count:** •

♠ **VALEUR-PRAGMA** ::= • •

Ce pragma n'a pas de valeur par défaut ; son nom ne peut donc pas figurer dans la liste des pragmas calculés d'un module.

A.3 Pragmas apparaissant uniquement dans les modules OC

A.3.1 Pragma "wire"

Le pragma "**wire**", associé à une entrée dans la table des variables, est utilisé pour indiquer que cette variable a été créée par le compilateur et qu'elle contient la valeur d'un fil dans la traduction d'un système d'équations en OC-monoboucle. Cela signifie en particulier que l'affectation d'une valeur à cette variable est valide dans l'instant présent. La syntaxe de ce pragma est la suivante :

♠ **Nom-pragma** ::= • **wire:** •

♠ **VALEUR-PRAGMA** ::= • •

Ce pragma n'a pas de valeur par défaut ; son nom ne peut donc pas figurer dans la liste des pragmas calculés d'un module.

A.3.2 Pragma "register"

Le pragma "**register**", associé à une entrée dans la table des variables, est utilisé pour indiquer que cette variable a été créée par le compilateur et qu'elle contient la valeur d'un registre dans la traduction d'un système d'équations en OC-monoboucle. Cela signifie en particulier que l'affectation d'une valeur à cette variable n'est valide qu'à l'instant suivant. La syntaxe de ce pragma est la suivante :

♠ **Nom-pragma** ::= • **register:** •

♠ **VALEUR-PRAGMA** ::= • •

Ce pragma n'a pas de valeur par défaut ; son nom ne peut donc pas figurer dans la liste des pragmas calculés d'un module.

A.3.3 Pragma "haltset"

Le pragma "**haltset**" est utilisé pour désigner un ensemble de **halts** (points d'arrêt). Sa syntaxe est la suivante :

♠ **Nom-pragma** ::= • **haltset:** •

♠ **VALEUR-PRAGMA** ::= • { **Index** ... } •

La valeur de ce pragma est une liste d'index désignant des pragmas dans la table des pragmas. Ces pragmas donnent des informations (ex. : référence source) associées à chaque `halt`. Le pragma "`haltset`" peut être associé à deux types d'objets :

- à un *dag* de la table des états : il indique alors l'ensemble des `halts` correspondant à cet état ;
- à une entrée de la table des variables : il indique alors un ensemble de `halts` actifs dans tous les états dans lesquels la variable, nécessairement de type booléen `$1`, a la valeur vraie `$2` (cas du OC-monoboucle).

Le nom de ce pragma peut figurer dans la liste des pragmas calculés d'un module et la valeur par défaut associée est la liste vide.

A.3.4 Pragma "emitted"

Le pragma "`emitted`" est utilisé pour indiquer un ensemble de signaux émis par l'automate dans une transition donnée. Sa syntaxe est la suivante :

♠ **Nom-pragma** ::= • **emitted:** •

♠ **VALEUR-PRAGMA** ::= • { **Index** ... } •

La valeur de ce pragma est une liste d'index désignant des signaux dans la table des signaux. Ce pragma peut être associé à deux types d'objets :

- à une instruction OC de changement d'état : il indique alors les signaux émis par l'automate lors de la transition de l'état courant vers le nouvel état en suivant cette branche du *dag*;
- à une entrée de la table des variables : il indique alors les signaux émis par l'automate lorsque la variable, nécessairement de type booléen `$1`, a la valeur vraie `$2` (cas du OC-monoboucle).

Le nom de ce pragma peut figurer dans la liste des pragmas calculés d'un module et la valeur par défaut associée est la liste vide.

A.3.5 Pragma "started"

Le pragma "`started`" est utilisé pour indiquer un ensemble de tâches démarrées par l'automate dans une situation donnée. Sa syntaxe est la suivante :

♠ **Nom-pragma** ::= • **started:** •

♠ **VALEUR-PRAGMA** ::= • { **Index** ... } •

La valeur de ce pragma est une liste d'index désignant des appels de tâches dans la table des **execs**. Ce pragma peut être associé à deux types d'objets :

- à une instruction OC de changement d'état : il indique alors les tâches démarrées par l'automate lors de la transition de l'état courant vers le nouvel état en suivant cette branche du *dag*;
- à une entrée de la table des variables : il indique alors les tâches démarrées par l'automate lorsque la variable, nécessairement de type booléen `$1`, a la valeur vraie `$2` (cas du OC-monoboucle).

Le nom de ce pragma peut figurer dans la liste des pragmas calculés d'un module et la valeur par défaut associée est la liste vide.

A.3.6 Pragma "killed"

Le pragma "**killed**" est utilisé pour indiquer un ensemble de tâches tuées par l'automate dans une situation donnée. Sa syntaxe est la suivante :

♠ **Nom-pragma** ::= • **killed:** •

♠ **VALEUR-PRAGMA** ::= • { **Index** ... } •

La valeur de ce pragma est une liste d'index désignant des appels de tâches dans la table des **execs**. Ce pragma peut être associé à deux types d'objets :

- à une instruction OC de changement d'état : il indique alors les tâches tuées par l'automate lors de la transition de l'état courant vers le nouvel état en suivant cette branche du *dag*;
- à une entrée de la table des variables : il indique alors les tâches tuées par l'automate lorsque la variable, nécessairement de type booléen `$1`, a la valeur vraie `$2` (cas du OC-monoboucle).

Le nom de ce pragma peut figurer dans la liste des pragmas calculés d'un module et la valeur par défaut associée est la liste vide.

A.3.7 Pragma "awaited"

Le pragma "**awaited**", associé à un *dag* de la table des états, est utilisé pour indiquer un ensemble de signaux attendus par l'automate dans cet état. Sa syntaxe est la suivante :

♠ **Nom-pragma** ::= • **awaited:** •

♠ **VALEUR-PRAGMA** ::= • { **Index** ... } •

La valeur de ce pragma est une liste d'index désignant des signaux d'entrée dans la table des signaux. Ces signaux sont tels que, si aucun d'eux n'est présent dans l'événement d'entrée, l'automate ne changera pas d'état. Plus exactement, cela signifie que tous les

signaux d'entrée qui n'apparaissent pas dans le pragma "**awaited**" ne peuvent pas faire changer l'automate d'état à eux seuls.

Le nom de ce pragma peut figurer dans la liste des pragmas calculés d'un module et la valeur par défaut associée est la liste des index de tous les signaux d'entrée (c'est-à-dire du type **input:**, **inputoutput:** ou **return:**) de l'automate.

Annexe B

Grammaire du format

B.1 CARACTÈRES

♠ Caractère ::= • caractère • CodeCaractère •

♠ caractère ::= • marque • délimiteur • séparateur • car-ident • autre-caractère •

♠ marque ::= • $ • : • . • + • - • # • @ • ! • ? • " • % •

♠ délimiteur ::=

• (•) • , • / • : • ; •
• < • > • [•] • { • } •

♠ séparateur ::= • *tabulation horizontale* • *nouvelle ligne* • *nouvelle page* •
• *retour chariot* • *espace* •

Caractères identificateurs

♠ car-ident ::= • car-lettre • car-chiffre • _ •

♠ car-lettre ::= • car-lettre-majuscule • car-lettre-minuscule •

♠ car-lettre-majuscule ::=

• A • B • C • D • E • F • G • H • I •
• J • K • L • M • N • O • P • Q • R •
• S • T • U • V • W • X • Y • Z •

♠ car-lettre-minuscule ::=

• a • b • c • d • e • f • g • h • i •
• j • k • l • m • n • o • p • q • r •
• s • t • u • v • w • x • y • z •

♠ car-chiffre ::= • 0 • 1 • 2 • 3 • 4 • 5 • 6 • 7 • 8 • 9 •

Caractères codés

♠ CodeCaractère ::= • CodeOctal • CodeHexadécimal • code-échappement •

♠ CodeOctal ::= • \ car-octal [car-octal [car-octal]] •

♠ car-octal ::= • 0 • 1 • 2 • 3 • 4 • 5 • 6 • 7 •

♠ CodeHexadécimal ::= • \x car-hexadécimal [car-hexadécimal] •

♠ car-hexadécimal ::=

• car-chiffre •
• A • B • C • D • E • F • a • b • c • d • e • f •

♠ code-échappement ::=

• \a • \b • \f • \n • \r • \t • \v • \\ • \" • \' • \? •

B.2 UNITÉS LEXICALES

Constantes entières

♠ Cst-entière ::= • { car-chiffre ... } •

Constantes réelles

♠ Cst-réelle ::= • Cst-réelle-simple-précision • Cst-réelle-double-précision •

♠ Cst-réelle-simple-précision ::= • Partie-fraction Exposant-simple-précision •

♠ Cst-réelle-double-précision ::= • Partie-fraction Exposant-double-précision •

♠ Partie-fraction ::= • Cst-entière . Cst-entière •

♠ Exposant-simple-précision ::= • e Cst-relative • E Cst-relative •

♠ Exposant-double-précision ::= • d Cst-relative • D Cst-relative •

♠ Cst-relative ::= • Cst-entière • + Cst-entière • - Cst-entière •

Constantes chaîne

♠ Cst-chaîne ::= • [”] [{ CaractèreChaîne ... }] [”] •

♠ CaractèreChaîne ::= • { Caractère ⊖ car-spec-chaîne } •

♠ car-spec-chaîne ::= • [”] • [\] •

Identificateurs

♠ Identificateur ::= • car-début-ident [{ car-ident ... }] •
• [$] car-début-ident [{ car-ident ... }] •

♠ car-début-ident ::= • { car-ident ⊖ car-chiffre } •

Index

♠ Index ::= • Index-local • Index-objet-prédéfini • Index-composé • Index-indéfini •

♠ Index-local ::= • Numéro-entrée •

♠ Numéro-entrée ::= • Cst-entière •

♠ Index-objet-prédéfini ::= • [$] Cst-entière • [$I] Cst-entière • [$g] Cst-entière •

♠ Index-composé ::= • Index-importation [.] Index-local •

♠ Index-importation ::= • Cst-entière •

♠ Index-indéfini ::= • [–] •

B.3 PAQUETS

♠ PAQUET ::=
• [package:] EN-TÊTE-PAQUET { ENTRÉE-PAQUET ... } [endpackage:] •

EN-TÊTE-PAQUET

♠ EN-TÊTE-PAQUET ::= • Nombre-entités Identificateur-*paquet* •

♠ Nombre-entités ::= • Cst-entière •

ENTRÉE-PAQUET

♠ ENTRÉE-PAQUET ::= • Numéro-entité [:] ENTITÉ •

♠ Numéro-entité ::= • Cst-entière •

♠ ENTITÉ ::= • BLOC-DE-DONNÉES •
• MODULE-IC •
• INTERFACE • NŒUD •
• MODULE-OC •

Méta description des tables

♠ TABLE-X ::= • [Nom-table-X] EN-TÊTE-TABLE-X
[{ ENTRÉE-TABLE-X ... }] [end:] •

♠ EN-TÊTE-TABLE-X ::= • Nombre-entrées [INFOS-TABLE-X] •

♠ Nombre-entrées ::= • Cst-entière •

♠ ENTRÉE-TABLE-X ::= • Numéro-entrée [:] OBJET-X [;] •

B.3.1 BLOCS DE DONNÉES

♠ BLOC-DE-DONNÉES ::= • [data:] INFOS-BLOC-DE-DONNÉES
[TABLE-IMPORTATIONS]
[TABLE-TYPES]
[TABLE-CONSTANTES]
[TABLE-FONCTIONS]
[TABLE-PROCÉDURES]
[TABLE-PRAGMAS]
[enddata:] •

♠ INFOS-BLOC-DE-DONNÉES ::=
• Format-dc Identificateur-*bloc-de-données* [[flat:]] •

♠ Format-dc ::= • [dc:] Numéro-version •

♠ Numéro-version ::= • Cst-entière •

B.3.2 MODULES IC

♠ MODULE-IC ::= • [module:] INFOS-MODULE-IC
PARTIE-COMMUNE-IMPÉRATIVE
TABLE-INSTRUCTIONS
[endmodule:] •

♠ INFOS-MODULE-IC ::= • Format-ic Identificateur-*module*
[flat:] [linked:]
[Propriété-de-processus] •

♠ Propriété-de-processus ::= • safe: • memorysafe: • unsafe: •

♠ Format-ic ::= • ic: Numéro-version •

♠ PARTIE-COMMUNE-IMPÉRATIVE ::= • [TABLE-IMPORTATIONS]
[TABLE-INSTANCES]
[TABLE-VARIABLES]
[TABLE-SIGNAUX]
[TABLE-RELATIONS]
[TABLE-TÂCHES]
[TABLE-APPELS-TÂCHES]
[TABLE-ACTIONS]
[TABLE-PRAGMAS] •

B.3.3 INTERFACES

♠ INTERFACE ::= • interface: INFOS-INTERFACE
[TABLE-IMPORTATIONS]
PARTIE-DÉCLARATIVE
endinterface: •

♠ INFOS-INTERFACE ::= • [Format-gc] Identificateur-*interface* [flat:] •

♠ Format-gc ::= • gc: Numéro-version •

♠ PARTIE-DÉCLARATIVE ::= • [TABLE-FLOTS]
[TABLE-ASSERTIONS]
[TABLE-SYNCHRONISATIONS]
[TABLE-DÉPENDANCES]
[TABLE-PRAGMAS] •

B.3.4 NŒUDS

♠ NŒUD ::= • NŒUD-LOCAL • NŒUD-EXTERNE •

NŒUD-LOCAL

♠ NŒUD-LOCAL ::= • node: INFOS-NŒUD-LOCAL
TABLE-IMPORTATIONS
PARTIE-DÉCLARATIVE
[TABLE-DÉFINITIONS]
endnode: •

♠ INFOS-NŒUD-LOCAL ::= • Format-gc Identificateur-*nœud* [**flat:**]
[Propriété-de-processus] •

NŒUD-EXTERNE

♠ NŒUD-EXTERNE ::= • **node:** INFOS-NŒUD-EXTERNE-IMPÉRATIF
TABLE-IMPORTATIONS
Index-*action*
[TABLE-PRAGMAS]
endnode: •
• **node:** INFOS-NŒUD-EXTERNE
TABLE-IMPORTATIONS
[TABLE-PRAGMAS]
endnode: •

♠ INFOS-NŒUD-EXTERNE-IMPÉRATIF ::=
• Format-gc Identificateur-*nœud* **ic:** [Propriété-de-processus] •

♠ INFOS-NŒUD-EXTERNE ::=
• Format-gc Identificateur-*nœud* **extern:** [Propriété-de-processus] •

B.3.5 MODULES OC

♠ MODULE-OC ::= • **module:** INFOS-MODULE-OC
PARTIE-COMMUNE-IMPÉRATIVE
[TABLE-DES-DAGS]
TABLE-DES-ÉTATS
endmodule: •

♠ INFOS-MODULE-OC ::= • Format-oc Identificateur-*module* [**flat:**]
[Propriété-de-processus] •

♠ Format-oc ::= • **oc:** Numéro-version •

B.4 TABLES

B.4.1 TABLE-IMPORTATIONS

♠ Nom-table-importations ::= • **import:** •

♠ Objet-importation ::= • Numéro-entité •
• Identificateur-*paquet* **.** Numéro-entité •

B.4.2 TABLE-TYPES

♠ Nom-table-types ::= • **types:** •

♠ OBJET-TYPE ::= • Identificateur-*type* • **$win** **(** Index-*type* **)** •

B.4.3 TABLE-CONSTANTES

♠ Nom-table-constantes ::= • **constants:** •

♠ OBJET-CONSTANTE ::=
• Identificateur-*constante* Index-*type* [VALEUR-CONSTANTE] •

♠ VALEUR-CONSTANTE ::= • **value:** EXPRESSION •

B.4.4 TABLE-FONCTIONS

♠ Nom-table-fonctions ::= • **functions:** •

♠ OBJET-FONCTION ::= • Identificateur-*fonction* **(** [{ Index-*type* **,** ... }] **)**
: Index-*type* [Propriété-de-processus] •

B.4.5 TABLE-PROCÉDURES

♠ Nom-table-procédures ::= • **procedures:** •

♠ OBJET-PROCÉDURE ::=
• Identificateur-*procédure* **(** [{ PARAMÈTRE **,** ... }] **)**
[Propriété-de-processus] •

♠ PARAMÈTRE ::= • PARAMÈTRE-ENTRÉE •
• PARAMÈTRE-SORTIE •
• PARAMÈTRE-ENTRÉE-SORTIE •

♠ PARAMÈTRE-ENTRÉE ::= • **i:** Index-*type* •

♠ PARAMÈTRE-SORTIE ::= • **o:** Index-*type* •

♠ PARAMÈTRE-ENTRÉE-SORTIE ::= • **io:** Index-*type* •

B.4.6 TABLE-PRAGMAS

- ♠ Nom-table-pragmas ::= • **pragmas:** •
- ♠ INFOS-TABLE-PRAGMAS ::= • **computed:** [{ Nom-pragma ... }] **;** •
- ♠ OBJET-PRAGMAS ::= • { Pragma ... } •
- ♠ Pragma ::= • Pragma-référence • Pragma-standard •
- ♠ Pragma-référence ::= • **%** Index-*pragmas* **%** •
- ♠ Pragma-standard ::= • **%** Nom-pragma Valeur-pragma **%** •
- ♠ Valeur-pragma ::= • [{ Unité-pragma ... }] •
- ♠ Unité-pragma ::= • CaractèrePragma • Cst-chaîne •
- ♠ CaractèrePragma ::= • { Caractère ⊖ car-spec-pragma } •
- ♠ car-spec-pragma ::= • **%** • car-spec-chaîne •

B.4.7 TABLE-INSTANCES

- ♠ Nom-table-instances ::= • **instances:** •
- ♠ INFOS-TABLE-INSTANCES ::= • **root:** Index-*instance* **;** •
- ♠ OBJET-INSTANCE ::= • Identificateur-*module* Index-*module-père* •

B.4.8 TABLE-VARIABLES

- ♠ Nom-table-variables ::= • **variables:** •
- ♠ OBJET-VARIABLE ::= • Index-*type* [VALEUR-INITIALE] •
- ♠ VALEUR-INITIALE ::= • **value:** EXPRESSION •

B.4.9 TABLE-SIGNAUX

♠ Nom-table-signaux ::= • **signals:** •

♠ OBJET-SIGNAL ::= • NATURE-DE-SIGNAL CANAL-DE-SIGNAL
[BOOL-DE-SIGNAL] •

♠ NATURE-DE-SIGNAL ::= • **input:** Identificateur-*signal* •
• **return:** Identificateur-*signal* •
• **output:** Identificateur-*signal* •
• **inputoutput:** Identificateur-*signal* •
• **local:** •

♠ CANAL-DE-SIGNAL ::=
• Index-*type* [VARIABLE-ASSOCIÉE] CARDINALITÉ •

♠ VARIABLE-ASSOCIÉE ::= • **value:** Index-*variable* •

♠ CARDINALITÉ ::= • **single:** • **multiple:** Index-*fonction* •

♠ BOOL-DE-SIGNAL ::= • **bool:** Index-*variable* •

B.4.10 TABLE-RELATIONS

♠ Nom-table-relations ::= • **relations:** •

♠ OBJET-RELATION ::= • EXCLUSION • IMPLICATION •

♠ EXCLUSION ::= • **(** Index-*signal* **,** { Index-*signal* **,** ... } **)** •

♠ IMPLICATION ::= • Index-*signal* Index-*signal* •

B.4.11 TABLE-TÂCHES

♠ Nom-table-tâches ::= • **tasks:** •

♠ OBJET-TÂCHE ::= • Identificateur-*tâche* **(** [{ PARAMÈTRE **,** ... }] **)**
[Propriété-de-processus] •

B.4.12 TABLE-APPELS-TÂCHES

♠ Nom-table-appels-tâches ::= • **execs:** •

♠ **OBJET-APPEL-TÂCHE** ::=

• Index-*tâche* Index-*signal* [( ] [{ **EXPRESSION** [,] ... }] [ )] •

B.4.13 TABLE-ACTIONS

♠ **Nom-table-actions** ::= • [**actions:**] •

♠ **OBJET-ACTION** ::= • { **ACTION-ÉLÉMENTAIRE** ... } •

♠ **ACTION-ÉLÉMENTAIRE** ::=

- • [**act:**] Index-*action* •
- • [**if:**] **EXPRESSION** •
- • [**call:**] Index-*procédure* [( ] [{ **EXPRESSION** [,] ... }] [ )] •
- • [**input:**] Index-*signal* •
- • [**output:**] Index-*signal* •
- • [**reset:**] Index-*variable* •
- • [**combine:**] Index-*signal* **EXPRESSION** •
- • [**start:**] Index-*exec* •
- • [**kill:**] Index-*exec* •
- • [**return:**] Index-*exec* •
- • [**suspend:**] Index-*exec* •
- • [**resume:**] Index-*exec* •

B.4.14 TABLE-INSTRUCTIONS

♠ **Nom-table-instructions** ::= • [**instructions:**] •

♠ **INFOS-TABLE-INSTRUCTIONS** ::= • [**startpoint:**] Index-*instruction* [;]
[**goloops:**] **Cst-entière** [;]
[**exitlevels:**] **Cst-entière** [;] •

♠ OBJET-INSTRUCTION ::=

- Action: [Index-*action*] (Index-*instruction*) •
- Test: [Index-*action*]
  (Index-*instruction* , Index-*instruction*) •
- Reset: [Index-*signal*] (Index-*instruction*) •
- Emit: [Index-*signal*] (Index-*instruction*) •
- Access: [Index-*signal*] (Index-*instruction*) •
- Goto: (Index-*instruction*) •
- Goloop: (Index-*instruction*) •
- Fork: { Index-*instruction* } ({ Index-*instruction* , ... }) •
- Parallel: ({ Index-*instruction* , ... }) < Index-*instruction* > •
- Exit: { Index-*instruction* } Cst-entière •
- Halt: < Index-*instruction* >
  [{ { Index-*appel-tâche* , ... } }] Cst-entière •
- Watchdog: { Index-*instruction* } < Index-*instruction* > •
- Stay: < Index-*instruction* > •
- Run: Identificateur-*module* [RENOMMAGES-IC]
  < Index-*instruction* > (Index-*instruction*) •
- Return: Cst-entière •

♠ RENOMMAGES-IC ::= • RENOMMAGE-TYPES-IC ;
RENOMMAGE-CONSTANTES-IC ;
RENOMMAGE-FONCTIONS-IC ;
RENOMMAGE-PROCÉDURES-IC ;
RENOMMAGE-TÂCHES-IC ;
RENOMMAGE-SIGNAUX-IC ;
Index-*dernier-signal* •

♠ RENOMMAGE-TYPES-IC ::= • { RENOMMAGE-TYPE-IC , ... } •

♠ RENOMMAGE-CONSTANTES-IC ::=

- { RENOMMAGE-CONSTANTE-IC , ... } •

♠ RENOMMAGE-FONCTIONS-IC ::=

- { RENOMMAGE-FONCTION-IC , ... } •

♠ RENOMMAGE-PROCÉDURES-IC ::=

- { RENOMMAGE-PROCÉDURE-IC , ... } •

♠ RENOMMAGE-TÂCHES-IC ::= • { RENOMMAGE-TÂCHE-IC , ... } •

♠ RENOMMAGE-SIGNAUX-IC ::= • { RENOMMAGE-SIGNAL-IC [,] ... } •

♠ RENOMMAGE-TYPE-IC ::= • Identificateur-*type* [/] Index-*type* •

♠ RENOMMAGE-CONSTANTE-IC ::=
• Identificateur-*constante* [/] Index-*constante* •

♠ RENOMMAGE-FONCTION-IC ::= • Identificateur-*fonction* [/] Index-*fonction* •

♠ RENOMMAGE-PROCÉDURE-IC ::=
• Identificateur-*procédure* [/] Index-*procédure* •

♠ RENOMMAGE-TÂCHE-IC ::= • Identificateur-*tâche* [/] Index-*tâche* •

♠ RENOMMAGE-SIGNAL-IC ::= • Identificateur-*signal* [/] Index-*signal* •

B.4.15 TABLE-FLOTS

♠ Nom-table-flots ::= • [flows:] •

♠ OBJET-FLOT ::= • NATURE-DE-FLOT CANAL-DE-FLOT Horloge •

♠ NATURE-DE-FLOT ::= • [i:] Identificateur-*flot* • [[o:]] Identificateur-*flot* •

♠ CANAL-DE-FLOT ::= • Index-*type* [VALEUR-INITIALE-FLOT] •

♠ VALEUR-INITIALE-FLOT ::= • [value:] EXPRESSION •

♠ Horloge ::= • Index-*flot* •

B.4.16 TABLE-ASSERTIONS

♠ Nom-table-assertions ::= • [assertions:] •

♠ OBJET-ASSERTION ::= • EXPRESSION • PARTIE-ÉQUATION •

♠ PARTIE-ÉQUATION ::= • ÉQUATION • INSTANCIATION •
• APPEL-PROCÉDURE •

ÉQUATION

♠ ÉQUATION ::= • define: Index-*flot* EXPRESSION •

INSTANCIATION

♠ INSTANCIATION ::=

• set: [Horloge] Index-*nœud* (RENOMMAGES) •

♠ RENOMMAGES ::= • [RENOMMAGE-TYPES]
[RENOMMAGE-CONSTANTES]
[RENOMMAGE-FONCTIONS]
[RENOMMAGE-PROCÉDURES]
RENOMMAGE-FLOTS •

♠ RENOMMAGE-TYPES ::= • types: { Index-*type* / Index-*type* , ... } •

♠ RENOMMAGE-CONSTANTES ::=

• constants: { Index-*constante* / Index-*constante* , ... } •

♠ RENOMMAGE-FONCTIONS ::=

• functions: { Index-*fonction* / Index-*fonction* , ... } •

♠ RENOMMAGE-PROCÉDURES ::=

• procedures: { Index-*procédure* / Index-*procédure* , ... } •

♠ RENOMMAGE-FLOTS ::= • flows: { EXPRESSION / Index-*flot* , ... } •

APPEL-PROCÉDURE

♠ APPEL-PROCÉDURE ::=

• call: [Horloge] Index-*procédure* ([{ EXPRESSION , ... }]) •

B.4.17 TABLE-SYNCHRONISATIONS

♠ Nom-table-synchronisations ::= • synchronizations: •

♠ OBJET-SYNCHRONISATION ::= • OBJET-ASSERTION •

B.4.18 TABLE-DÉPENDANCES

♠ Nom-table-dépendances ::= • dependences: •

♠ OBJET-DÉPENDANCE ::=

• LISTE-INDEX-FLOTS LISTE-INDEX-FLOTS [Horloge] •

♠ LISTE-INDEX-FLOTS ::= • Index-*flot* • [( ] { Index-*flot* [,] ... } [ )] •

B.4.19 TABLE-DÉFINITIONS

♠ Nom-table-définitions ::= • [definitions:] •

♠ OBJET-DÉFINITION ::= • PARTIE-ÉQUATION • ACTIVATION •
• DÉPENDANCES-DÉFINITIONS •

♠ ACTIVATION ::=
• [do:] [Horloge] Index-*nœud* [( ] [{ EXPRESSION [,] ... }] [ )] •

♠ DÉPENDANCES-DÉFINITIONS ::=
• LISTE-INDEX-DÉFINITIONS LISTE-INDEX-DÉFINITIONS
[Horloge] •

♠ LISTE-INDEX-DÉFINITIONS ::= • Index-*définition* •
• [( ] { Index-*définition* [,] ... } [ )] •

B.4.20 TABLE-DAGS

♠ Nom-table-dags ::= • [dags:] •

♠ INFOS-TABLE-DAGS ::= • [calls:] Cst-entière [;] •

♠ OBJET-DAG ::= • DAG-FERMÉ •
• DAG-OUVERT •

♠ DAG-FERMÉ ::= • DAG-OUVERT FERMETURE-DE-DAG •

♠ DAG-OUVERT ::= • [LISTE-DE-DAGS] [{ Index-*action-pure* ... }] •

♠ LISTE-DE-DAGS ::= • { ACTIONS-ET-TEST-OUVERT ... } •

♠ ACTIONS-ET-TEST-OUVERT ::= • { Index-*action-pure* ... } TEST-OUVERT •

♠ FERMETURE-DE-DAG ::= • [<] Index-*état* [>] •
• [[ ] Index-*dag* [ ]] •
• TEST-FERMÉ •

♠ TEST-FERMÉ ::=
• Index-*action-de-test* [( ] DAG-FERMÉ [ )] [( ] DAG-FERMÉ [ )] •

♠ TEST-OUVERT ::=

- • [{] Index-*dag* [}] •
- • Index-*action-de-test* [( ] DAG-OUVERT [ )] [( ] DAG-FERMÉ [ )] •
- • Index-*action-de-test* [( ] DAG-FERMÉ [ )] [( ] DAG-OUVERT [ )] •
- • Index-*action-de-test* [( ] DAG-OUVERT [ )] [( ] DAG-OUVERT [ )] •

B.4.21 TABLE-ÉTATS

♠ Nom-table-états ::= • [states:] •

♠ INFOS-TABLE-ÉTATS ::= • [startpoint:] Index-*état-de-départ* [;]
[[sink:] Index-*état-puits* [;]]
[calls:] Cst-entière [;] •

♠ OBJET-ÉTAT ::= • DAG-FERMÉ •

B.5 EXPRESSIONS

♠ EXPRESSION ::= • APPEL-FONCTION • Terme •

♠ APPEL-FONCTION ::=

- • Index-*fonction* [( ] [{ EXPRESSION [,] ... }] [ )] •

♠ Terme ::= • Atome • RefConstante • Terme-IC-OC • Terme-GC •

♠ Atome ::= • [#] Cst-chaîne • [#] Cst-entière • [#] Cst-réelle •

♠ RefConstante ::= • [@] Index-*constante* •

Terme-IC-OC

♠ Terme-IC-OC ::= • Présence-signal • Valeur-signal • Index-*variable* •

♠ Présence-signal ::= • [!] Index-*signal* •

♠ Valeur-signal ::= • [?] Index-*signal* •

Terme-GC

♠ Terme-GC ::= • Index-*flot* •

Annexe C

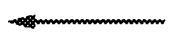
Une sémantique opérationnelle du code IC

C.1 Exemples d'exécution des instructions IC par le processeur LCOC

Le processeur LCOC du compilateur ESTEREL traduit les programmes IC `linked:` en automates finis déterministes. Rappelons que le code IC avec l'attribut `linked:` est du code IC ne contenant plus d'instruction `Run`. Les exemples présentés dans cette section sont basés sur l'algorithme d'exécution du processeur LCOC version 3. D'autres exécutions sont possibles sans faire référence à une évolution dynamique des chemins de contrôle.

Nous présentons dans cette section des exemples d'exécution des programmes IC. L'exécution du IC est reliée à la notion d'instant (transition) d'un programme ESTEREL : un instant est une *exécution maximale* d'instructions IC. Une exécution est dite maximale lorsque le contrôle est arrêté sur des `Halt` ou `Return` (appelés *points d'arrêt*). Après une exécution, nous avons une *reconstruction* qui prépare le programme IC pour l'exécution suivante. La reconstruction ordonne les instructions qui seront exécutées dans l'instant suivant et calcule le nouveau point de départ du programme. Cette phase nécessite l'introduction d'une mémoire associée aux `Watchdog` et aux `Parallel` et qui sera matérialisée dans nos exemples par une boîte à côté de l'instruction concernée. Une flèche à côté d'un point d'arrêt indique que le contrôle est bloqué sur cette instruction. A coté du code IC, nous donnons une représentation graphique de celui-ci. Nous trouverons les connexions suivantes :

Reconstruction


Contrôle statique

Contrôle dynamique

Les connexions de reconstruction sont statiques et représentent les champs entre piquants ("<" et ">") des instructions IC. Les chemins de contrôle statiques visualisent les continuations des instructions et les chemins de contrôle dynamiques représentent les mémoires introduites précédemment. Les instructions associées comme `Watchdog/Test` ou `Parallel/Fork` par exemple sont représentées par deux boîtes côte à côte.

Nous donnons ici une exécution abstraite des instructions IC qui permet de bien mettre en évidence les deux étapes d'une exécution ainsi que la sémantique des ins-

tructions : l'ordre d'exécution des instructions dans deux branches d'un **parallel** est arbitraire. Nous ne réglons donc aucun des problèmes comme les dialogues instantanés ou les cycles de causalité. Un calcul de l'ordre d'exécution est donné dans la thèse de G. Gonthier ; il se base sur le calcul des signaux potentiellement émis par une instruction IC.

Exemple 1 : Le programme suivant :

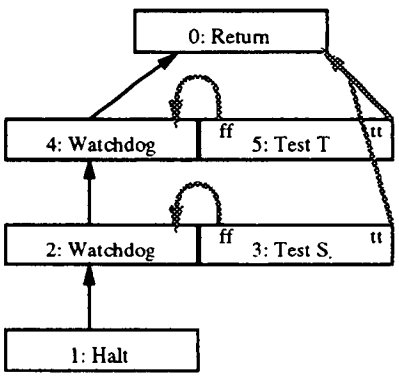
```
do
   await S
watching T
```

se traduit en IC de la façon suivante :

```
...
signals: 2
0: input: T $0 multiple: - bool: 0;
1: input: S $0 multiple: - bool: 1;
end:
...
statements: 6 startpoint: 1;
goloops: 0;
exitlevels: 0;
0: Return: 0;
1: Halt: <2> 1;
2: Watchdog: {3} <4>;
3: Test: [1] (0,2);
4: Watchdog: {5} <0>;
5: Test: [0] (0,4);
end:
endmodule:
```

Reprenons les informations nécessaires à l'exécution de ce module en remplaçant les index de signaux par leurs noms. Dans la représentation graphique, seules les connexions statiques sont figurées (reconstruction, instructions associées et contrôle statique).

```
        startpoint: 1
        0: Return: 0
        1: Halt: <2> 1
  -     2: Watchdog: {3} <4>
        3: Test: S (0,2)
  -     4: Watchdog: {5} <0>
        5: Test: T (0,4)
```



L'exécution commence à l'instruction dont l'index est donné par **startpoint** : dans notre exemple, nous trouvons le **Halt** d'index 1. Le contrôle est donc bloqué et l'exécution maximale se termine.

La reconstruction commence à partir de ce **Halt** en appliquant les règles suivantes :

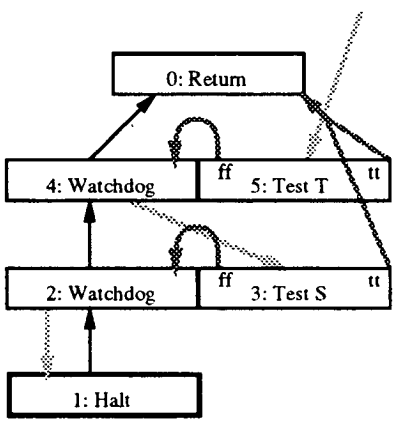
- l'index d'un `Halt` se met dans la mémoire de son père,
- l'index d'un `Parallel` se met dans la mémoire de son père,
- le champ *garde* d'un `Watchdog` se met dans la mémoire de son père,
- lorsque le père est un `Return`, la reconstruction se termine et l'index remonté devient le nouveau point d'entrée.

Pour la représentation graphique, ces règles deviennent :

- ajouter un chemin de contrôle dynamique entre l'instruction père (suivre le chemin de reconstruction) d'un `Halt` et ce `Halt`,
- ajouter un chemin de contrôle dynamique entre l'instruction père d'un `Parallel` et ce `Parallel`,
- ajouter un chemin de contrôle dynamique entre l'instruction père d'un `Watchdog` et l'instruction associée de ce `Watchdog`.
- lorsque le père est un `Return`, la reconstruction se termine et l'instruction pointée par le dernier chemin dynamique sera le nouveau point d'entrée.

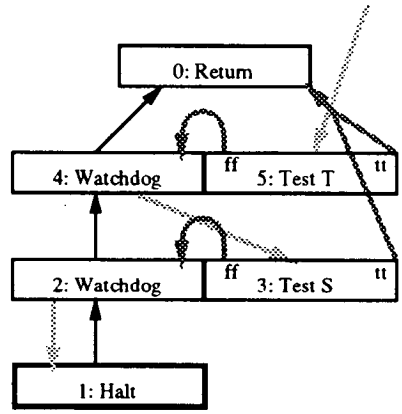
On obtient ainsi un nouveau programme avec le contrôle bloqué sur le `Halt` 1. Dans la représentation graphique, on montre le chemin de contrôle dynamique et le `Halt` 1 est encadré en gras.

```
        startpoint: 5
        0: Return: 0
   →    1: Halt: <2> 1
 [1]    2: Watchdog: {3} <4>
        3: Test: S (0,2)
 [3]    4: Watchdog: {5} <0>
        5: Test: T (0,4)
```



L'exécution du deuxième instant commence à l'instruction donnée par la reconstruction qui précède : le `Test` d'index 5. Si nous supposons qu'aucune entrée n'est présente, l'exécution continue sur le `Watchdog` 4. Pour exécuter un `Watchdog`, nous exécutons l'instruction contenue dans sa mémoire : ici le `Test` 3. Puis finalement le `Watchdog` 2 et le `Halt`. L'exécution sur la représentation graphique se fait simplement en suivant les chemins de contrôle dynamique ou statique. Il faut aussi faire les choix appropriés dans les instructions de test (comme les `Test` ici). La reconstruction est la même que pour le premier instant :

```
        startpoint: 5
        0: Return: 0
   →    1: Halt: <2> 1
[1]     2: Watchdog: {3} <4>
        3: Test: S (0,2)
[3]     4: Watchdog: {5} <0>
        5: Test: T (0,4)
```



Si nous supposons maintenant que S ou T sont présents, l'exécution s'arrête sur le `Return` et le programme est alors terminé.

Exemple 2 : Le programme suivant :

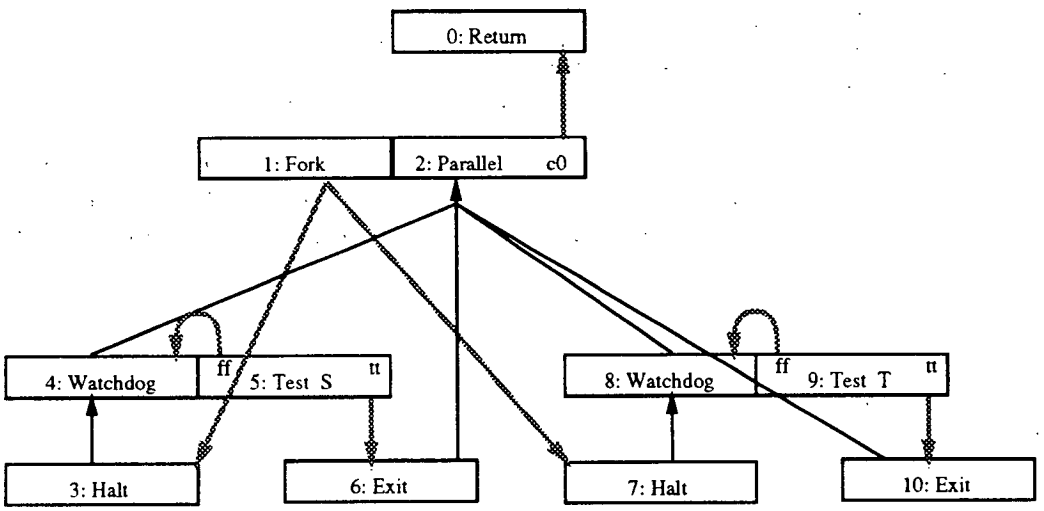
```
   await S
||
   await T
```

se traduit en IC de la façon suivante :

```
...
signals: 2
0: input: T $0 multiple: - bool: 0;
1: input: S $0 multiple: - bool: 1;
end:
...
statements: 11 startpoint: 1;
goloops: 0;
exitlevels: 0;
0: Return: 0;
1: Fork: {2} (3, 7);
2: Parallel: (0) <0>;
3: Halt: <4> 1;
4: Watchdog: {5} <2>;
5: Test: [1] (6,4);
6: Exit: {2} 0;
7: Halt: <8> 2;
8: Watchdog: {9} <2>1;
9: Test: [0] (10,8);
10: Exit {2} 10;
end:
endmodule:
```

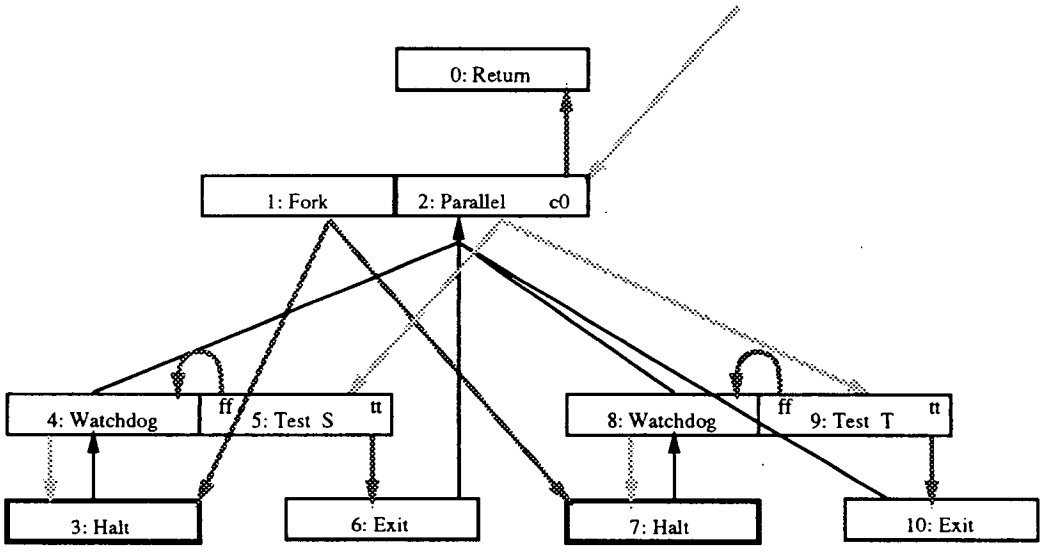
Le programme initial est le suivant :

```
        startpoint: 1
        0: Return: 0
        1: Fork: {2} (3, 7)
  [ - ] 2: Parallel: (0) <0>
        3: Halt: <4> 1
  [ - ] 4: Watchdog: {5} <2>
        5: Test: S (6,4)
        6: Exit: {2} 0
        7: Halt: <8> 2
  [ - ] 8: Watchdog: {9} <2>
        9: Test: T (10,8)
        10: Exit: {2} 0
```



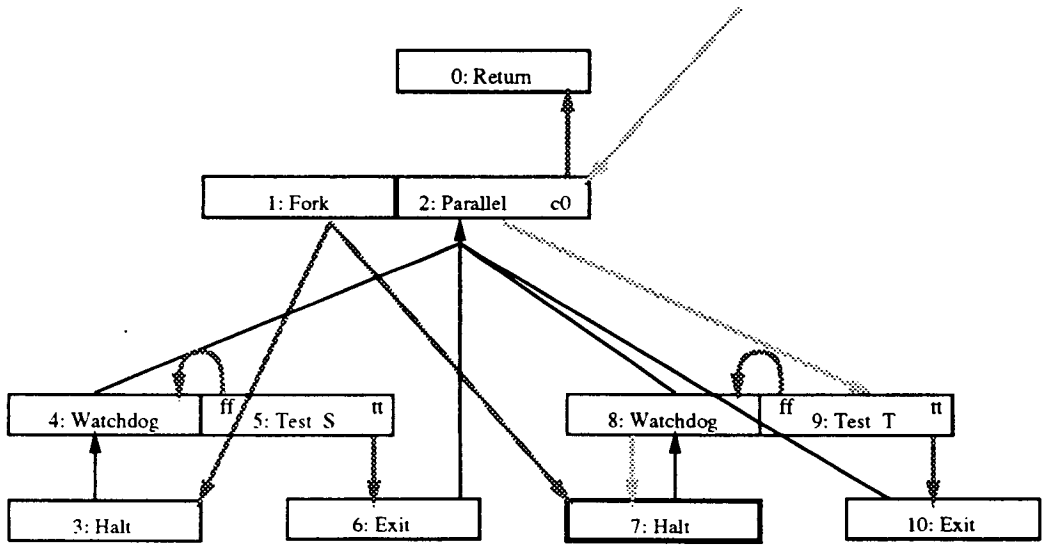
Au premier instant, on exécute le **Fork** 1 puis en parallèle les deux **Halt** 3 et 7. Ces deux **Halt** servent de base à la reconstruction qui donne :

```
        startpoint: 2
        0: Return: 0
        1: Fork: {2} (3, 7)
 [5, 9] 2: Parallel: (0) <0>
    →   3: Halt: <4> 1
  [ 3 ] 4: Watchdog: {5} <2>
        5: Test: S (6,4)
        6: Exit: {2} 0
    →   7: Halt: <8> 2
  [ 7 ] 8: Watchdog: {9} <2>
        9: Test: T (10,8)
        10: Exit: {2} 0
```



Au deuxième instant si S est présent : pour exécuter le **Parallel** 2, on exécute les instructions contenues dans sa mémoire. On aboutit à l'**Exit** 6 et au **Halt** 7. Le niveau de sortie du **Parallel** est donc $1 = max(0, 1)$. La reconstruction part du seul **Halt** actif et donne :

```
        startpoint: 2
        0: Return: 0
        1: Fork: {2} (3, 7)
[9]     2: Parallel: (0) <0>
        3: Halt: <4> 1
[-]     4: Watchdog: {5} <2>
        5: Test: S (6,4)
        6: Exit: {2} 0
   →    7: Halt: <8> 2
[7]     8: Watchdog: {9} <2>
        9: Test: T (10,8)
        10: Exit: {2} 0
```



Tant que T n'est pas présent, on obtient toujours le même programme. Si T est présent, on exécute le **Parallel** 2 puis le **Test** 9 et donc l'**Exit** 10. Le niveau de sortie du **Parallel**

est alors 0 : il se continue sur le **Return** et l'exécution du programme est terminée :

```
        startpoint: 0
        0: Return: 0
        1: Fork: {2} (3, 7)
  -     2: Parallel: (0) <0>
        3: Halt: <4> 1
  -     4: Watchdog: {5} <2>
        5: Test: S (6,4)
        6: Exit: {2} 0
  →     7: Halt: <8> 2
  -     8: Watchdog: {9} <2>
        9: Test: T (10,8)
        10: Exit: {2} 0
```

Exemple 3 : Le programme suivant :

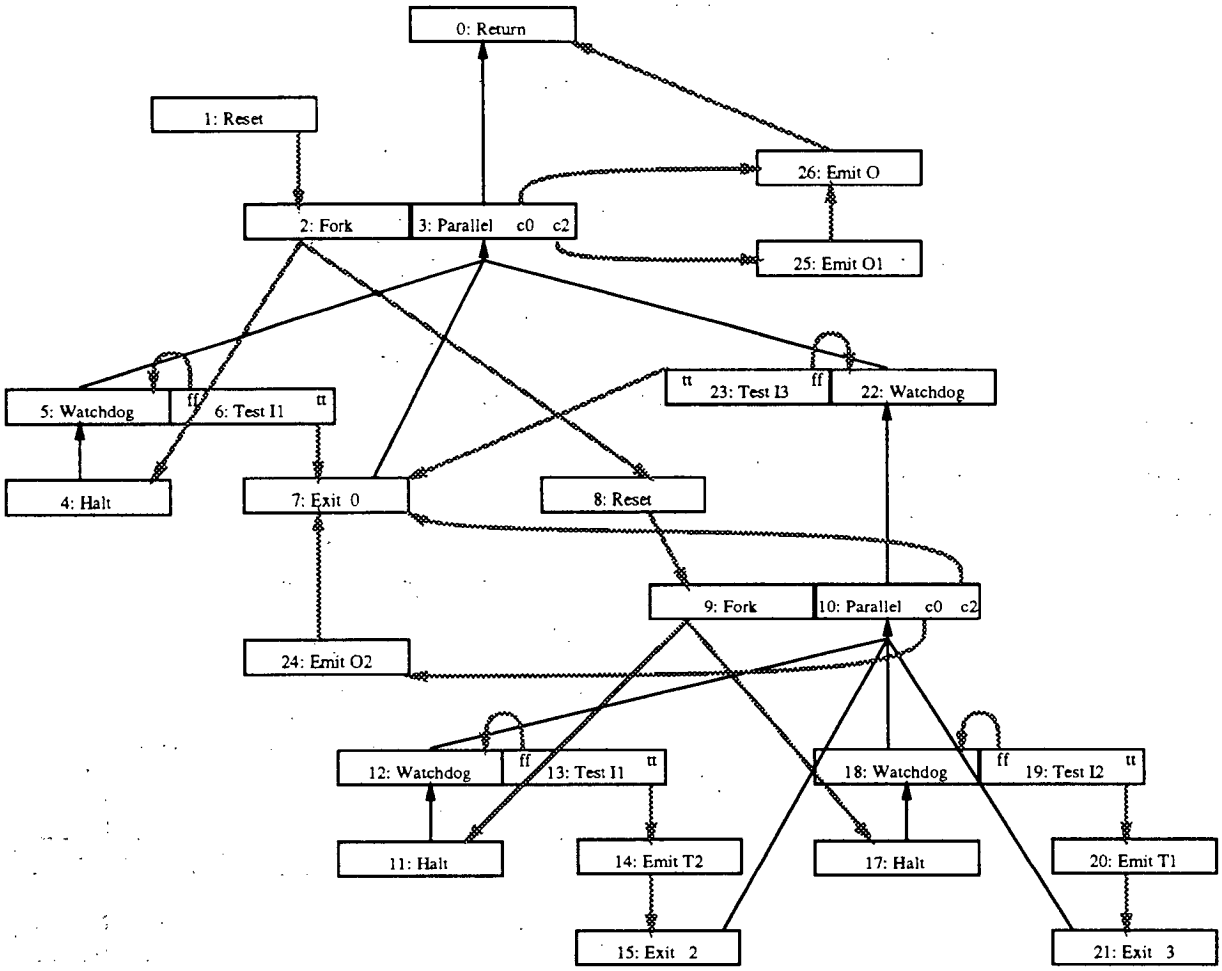
```
input I1, I2, I3;
output O, O1, O2;

trap T1 in
   [
      await I1
   ||
      trap T2 in
         do
            [
               await I1; exit T2
            ||
               await I2; exit T1
            ]
         watching I3
      handle T2 do
         emit O2
      end trap;
   ]
handle T1 do
   emit O1
end trap;
emit O
```

se traduit en IC de la façon suivante :

```
...
signals: 8
0: input: I1 $0 multiple: - bool: 0
1: input: I2 $0 multiple: - bool: 1
2: input: I3 $0 multiple: - bool: 2
3: output: 0 $0 multiple: - ;
4: output: 01 $0 multiple: - ;
5: output: 02 $0 multiple: - ;
6: local: $0 multiple: - ;
7: local: $0 multiple: - ;
end:
...
statements: 27 startpoint: 1;
goloops: 0;
exitlevels: 3;
0: Return: 0;
1: Reset: [6] (2);
2: Fork: {3} (4, 8);
3: Parallel: (26, 25) <0>;
4: Halt: <5> 1;
5: Watchdog: {6} <3>;
6: Test: [0] (7,5);
7: Exit: {3} 0;
8: Reset: [7] (9);
9: Fork: {10} (11, 17);
10: Parallel: (7, 24) <22>;
11: Halt: <12> 2;
12: Watchdog: {13} <10>;
13: Test: [0] (14,12);
14: Emit: [7] (15);
15: Exit: {10} 2;
16: Exit: {10} 0;
17: Halt: <18> 3;
18: Watchdog: {19} <10>;
19: Test: [1] (20,18);
20: Emit: [6] (21);
21: Exit: {10} 3;
22: Watchdog: {23} <3>;
23: Test: [2] (7,22);
24: Emit: [5] (7);
25: Emit: [4] (26);
26: Emit: [3] (0);
end:
```

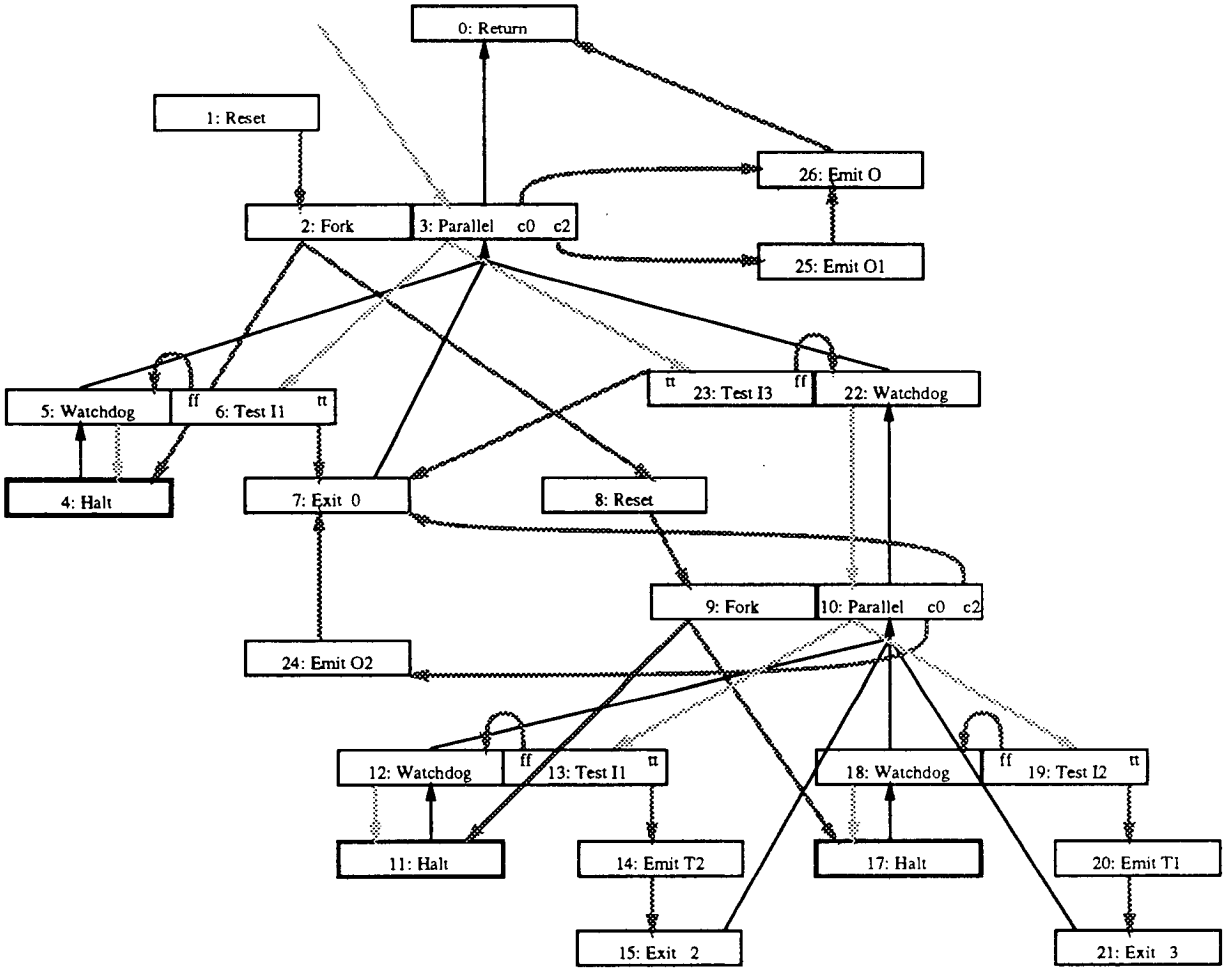
L'exécution commence à partir de l'état suivant :

```
        startpoint: 1
        0: Return: 0
        1: Reset: T1 (2)
        2: Fork: {3} (4, 8)
  -     3: Parallel: (26, 25) <0>
        4: Halt: <5> 1
  -     5: Watchdog: {6} <3>
        6: Test: I1 (7,5)
        7: Exit: {3} 0
        8: Reset: T2 (9)
        9: Fork: {10} (11, 17)
  -     10: Parallel: (7, 24) <22>
        11: Halt: <12> 2
  -     12: Watchdog: {13} <10>
        13: Test: I1 (14,12)
        14: Emit: T2 (15)
        15: Exit: {10} 2
        16: Exit: {10} 0
        17: Halt: <18> 3
  -     18: Watchdog: {19} <10>
        19: Test: I2 (20,18)
        20: Emit: T1 (21)
        21: Exit: {10} 3
  -     22: Watchdog: {23} <3>
        23: Test: I3 (7,22)
        24: Emit: 02 (7)
        25: Emit: 01 (26)
        26: Emit: 0 (0)
```



On est alors bloqué sur les `Halt` 4, 11 et 17 et après reconstruction nous obtenons :

```
                startpoint: 3
                0: Return: 0
                1: Reset: T1 (2)
                2: Fork: {3} (4, 8)
[6, 23]         3: Parallel: (26, 25) <0>
      →         4: Halt: <5> 1
[4]             5: Watchdog: {6} <3>
                6: Test: I1 (7,5)
                7: Exit: {3} 0
                8: Reset: T2 (9)
                9: Fork: {10} (11, 17)
[13, 19]        10: Parallel: (7, 24) <22>
      →         11: Halt: <12> 2
[11]            12: Watchdog: {13} <10>
                13: Test: I1 (14,12)
                14: Emit: T2 (15)
                15: Exit: {10} 2
                16: Exit: {10} 0
          →     17: Halt: <18> 3
[17]            18: Watchdog: {19} <10>
                19: Test: I2 (20,18)
                20: Emit: T1 (21)
                21: Exit: {10} 3
[10]            22: Watchdog: {23} <3>
                23: Test: I3 (7,22)
                24: Emit: O2 (7)
                25: Emit: O1 (26)
                26: Emit: O (0)
```



Pour le deuxième instant, nous nous intéressons uniquement au cas où I2 est présent pour montrer les interactions entre `Exit` et `Parallel`. On exécute donc `3: Parallel`, `23: Test`, `22: Watchdog` et `10: Parallel`. L'exécution de la première branche de ce dernier `Parallel` commence en 13 et aboutit à `11: Halt` ; la deuxième commence en `19: Test` et aboutit à `21: Exit` {10} 3. Le niveau de sortie de `10: Parallel` est donc 3. Or ce `Parallel` traite les niveaux de sortie jusqu'à 2 ; il transmet donc au `Parallel` père un niveau de sortie de $3 - 1 = 2$. Depuis `10: Parallel`, nous trouvons `22: Watchdog` et `3: Parallel` qui prend la continuation correspondant au niveau 2 soit `25: Emit: O1 (26)`. Ensuite, on exécute `26: Emit` et `0: Return` et l'exécution de l'instant est terminée. Toutes les reconstructions partiront désormais de `0: Return` : le programme est donc terminé.

Remarques :

Le calcul du niveau de sortie des `Parallel` peut être simplifié par un calcul lors de l'exécution des `Fork` :

> Lors de l'exécution d'un `Fork`, on ajoute à la fin de la liste de continuations du `Parallel` associé toutes les continuations de niveau supérieur ou égal à 2 du `Parallel` englobant.

Dans l'exemple précédent, on arrive sur `2: Fork` avec la liste vide et donc `3: Parallel` n'est pas modifié. Nous retenons simplement la nouvelle liste de continuation soit `(25)` (la seule continuation de niveau supérieur ou égale à 2). Lors de l'exécution de `9: Fork`,

nous ajoutons cette liste à `10: Parallel` pour obtenir `10: Parallel: (7, 24, 25) <22>`. La nouvelle liste est `(24, 25)` et `10: Parallel` peut maintenant traiter les `Exit` jusqu'au niveau 3.

C.2 Une sémantique opérationnelle du code IC

Dans cette section, nous définissons une sémantique opérationnelle du code IC à l'aide d'une machine abstraite. Nous voulons mettre en évidence la mécanique d'exécution des instructions IC indépendamment du langage ESTEREL pour donner à IC un statut de langage à part entière et permettre de l'utiliser dans d'autres contextes. Cette mécanique d'exécution est celle utilisée par le processeur LCOC version 3.

La reconstruction présentée dans les exemples et formalisée dans la suite calcule *dynamiquement* un grand nombre de données. En fait ces données calculées dynamiquement sont toujours des sous-ensemble de données *statiquement* calculable à partir d'un code IC. D'autres algorithmes d'exécution du IC existent, en particulier celui utilisé dans ESTEREL V4.

Nous donnons en premier lieu quelques définitions des types et des fonctions utilisés ensuite dans la sémantique.

C.2.1 Définitions

Dans la suite, *Objet* (avec une majuscule) désigne un ensemble d'*objets* (avec une minuscule). Nous introduisons les notations suivantes :

- Les instructions

  instruction dénote une instruction IC.

  instrIndexListe dénote une liste d'index d'instructions.

- Les points d'arrêt

  haltset est un ensemble d'index de points d'arrêt.

  niveau est un entier donnant le niveau de sortie d'une instruction.

- Les mémoires

  memInstr est un tableau associant à un index d'instruction une liste d'instructions à exécuter que nous appellerons dans la suite *instructions associées*. Cette liste est non vide pour les `Parallel` et les `Watchdog`. Elle correspond à la mémoire associée dans les exemples précédents.

  memPar est un tableau associant à un index d'instruction un triplet contenant :

  - un ensemble d'index de points d'arrêt. Cet ensemble est non vide pour les `Parallel` et donne les index des `Halt` sur lequel un `Parallel` est bloqué. Cet ensemble sera noté $M[p].haltset$.
  - un compteur donnant le nombre de branches actives d'un `Parallel` (i.e. les branches pour lesquelles le contrôle n'est ni sur un `Halt` ni sur un `Exit`) noté $M[p].active$.
  - un entier donnant le niveau de sortie d'un `Parallel` noté $M[p].niveau$.

Avec les définitions précédentes et pour un programme P donné, nous définissons les fonctions :

- `Executer`$_P$:
  $Instruction \times InstrIndexListe \times MemInstr \times MemPar \times Haltset$
  $\longrightarrow InstrIndexListe \times MemPar \times Haltset$

  Cette fonction exécute l'instruction en argument en utilisant des informations contenues dans $memInstr$. Elle modifie $memPar$ et $instrIndexListe$ qui donne les index des instructions restant à exécuter.

- `TrouverContinuation`$_P$:
  $Instruction \times Niveau \times Haltset \times InstrIndexListe \times MemPar \times Haltset$
  $\longrightarrow InstrIndexListe \times MemPar \times Haltset$

  Cette fonction est utilisée par `Executer`$_P$ pour calculer les continuations des instructions `Parallel`. Elle modifie la mémoire des `Parallel` et rend une liste d'index d'instructions restant à exécuter.

- `Reconstruire`$_P$: $Haltset \longrightarrow MemPar \times InstrIndexListe$

  Cette fonction calcule la mémoire des `Parallel` et des `Watchdog` ainsi que le ou les nouveaux points d'entrée. La reconstruction part de tous les points d'arrêt sur lesquels le contrôle est bloqué après l'exécution.

- `Reconst`$_P$:
  $Instruction \times Index \times MemInstr \times InstrIndexListe$
  $\longrightarrow MemInstr \times InstrIndexListe$

  Cette fonction auxiliaire est utilisée par `Reconstruire`$_P$.

Notations :

Pour un tableau M, nous noterons $M[p \leftarrow val]$ un nouveau tableau égal à M sauf pour la valeur correspondant à p qui devient val.

$M_\perp$ désigne un tableau dont toutes les entrées sont non initialisées.

Pour terminer, () note la liste vide, $+$ la concaténation de deux listes et $\perp$ désigne une valeur non initialisée.

C.2.2 La sémantique

Nous définissons :

- $R \in InstrIndexListe$ est l'ensemble des instructions restant à exécuter,
- $M_{instr} \in MemInstr$ est la mémoire des instructions,
- $M_{par} \in MemPar$ est la mémoire des `Parallel`,
- $H \in Haltset$ est l'ensemble des points d'arrêt du programme,
- $Start \in InstrIndexListe$ est l'ensemble des points d'entrée pour l'exécution de l'instant suivant.

Nous définissons l'exécution d'un programme par l'algorithme suivant :

- Initialisation au premier instant :
  - L'instruction initiale `startpoint` est celle donnée par le programme IC
    $Start = (\texttt{startpoint})$
  - La mémoire des instructions est non initialisée
    $M_{instr} = M_{\perp}$

- À chaque instant :

  1 Exécution maximale :

  1.1 $R = Start$, $M_{par} = M_{\perp}$ et $H = ()$

  1.2 tant que $R = (c_1, \ldots, c_n)$ est non vide, choisir i, $1 \leq i \leq n$ et faire
  $R, M_{par}, H = Executer_{\texttt{P}}\ \texttt{P}[c_i], (c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n), M_{instr}, M_{par}, H$

  2 Reconstruction :
  $M_{instr}, Start = Reconstruire_{\texttt{P}}\ H$

L'exécution d'un instant est alors terminée et le programme est prêt pour l'instant suivant.

Définition de Executer$_{\texttt{P}}$

Nous définissons `Executer`$_{\texttt{P}}$ en fonction du type d'instruction :

- Un `Return` ne fait rien :
  $\texttt{Executer}_{\texttt{P}}\ [\![\texttt{n} : \texttt{Return} : \texttt{r}]\!], R, M_{instr}, M_{par}, H = R, M_{par}, H$

- `Action` ajoute sa continuation aux instructions restant à exécuter (nous ne nous préoccupons pas de l'effet de l'action sur l'extérieur) :
  $\texttt{Executer}_{\texttt{P}}\ [\![\texttt{n} : \texttt{Action} : [\texttt{k}]\ (\texttt{c})]\!], R, M_{instr}, M_{par}, H = R + (\texttt{c}), M_{par}, H.$

- Un `Watchdog` ajoute ses instructions associées à celles restant à exécuter :
  $\texttt{Executer}_{\texttt{P}}\ [\![\texttt{n} : \texttt{Watchdog} : \{\texttt{r}\} < \texttt{f} >]\!], R, M_{instr}, M_{par}, H =$
  $R + M_{instr}[\texttt{n}], M_{par}, H$

- Un `Fork` ajoute aux instructions restant à exécuter les instructions qu'il lance en parallèle et initialise la mémoire du `Parallel` qui lui est associé :
  $\texttt{Executer}_{\texttt{P}}\ [\![\texttt{n} : \texttt{Fork} : \{\texttt{p}\}(\texttt{c}_1, \ldots, \texttt{c}_\texttt{m})]\!], R, M_{instr}, M_{par}, H =$
  $R + (c_1, \ldots, c_m), M_{par}[\texttt{p} \leftarrow (\emptyset, \texttt{m}, 0)], H$

  Remarque : dans M_{par}, le niveau de sortie de l'instruction `Parallel` est mis à 0. On pourrait aussi l'initialiser avec un entier négatif ou nul. En effet, les niveaux de sortie donnés par les `Exit` et les `Halt` sont des entiers positifs ou nuls et le calcul du niveau de sortie d'une instruction `Parallel` se fait sur le maximun des niveaux de sortie de ces fils. Cette remarque vaut aussi pour l'instruction `Parallel` qui suit.

- Un `Parallel` ajoute ses instructions associées à celles restant à exécuter et initialise le triplet qui lui correspond dans la mémoire des `Parallel` :

  $\texttt{Executer}_P\ [\![\texttt{n} : \texttt{Parallel} : (\texttt{c}_0, \ldots, \texttt{c}_m) < \texttt{f} >]\!], R, M_{instr}, M_{par}, H =$
  $R + M_{instr}[\texttt{n}], M_{par}[\texttt{n} \leftarrow (\emptyset, k, 0)], H$
  où k est le nombre d'éléments dans $M_{instr}[\texttt{n}]$

  Remarque : k est le nombre d'instructions exécutées par le `Parallel`.

- Un `Exit` transmet son niveau de sortie au `Parallel` père :

  $\texttt{Executer}_P\ [\![\texttt{n} : \texttt{Exit} : \{\texttt{p}\}\ \texttt{e}]\!], R, M_{instr}, M_{par}, H =$
  $\texttt{TrouverContinuation}_P\ \texttt{P[p]}, \texttt{e}, \emptyset, R, M_{par}, H$

- Un `Halt` transmet son niveau de sortie et son index. L'index du `Halt` sert pour le calcul de l'ensemble des points d'arrêt qui sera utilisé lors de la reconstruction :

  $\texttt{Executer}_P\ [\![\texttt{n} : \texttt{Halt} :< \texttt{f} >]\!], R, M_{instr}, M_{par}, H =$
  $\texttt{TrouverContinuation}_P\ \texttt{P[f]}, 1, \{\texttt{n}\}, R, M_{par}, H$

Remarque pour ESTEREL :
La définition de l'exécution des autres instructions du IC n'est pas présentée ici. Elle ajouterait des difficultés que nous ignorons volontairement dans un souci de simplification. Par exemple dans le cas du IC généré par ESTEREL, nous ne traitons pas les problèmes de causalité. Une résolution de ces problèmes qui permet de lever le non-déterminisme du pas 1.2 de l'exécution est proposée dans la thèse de G. Gonthier par un calcul des signaux potentiellement émis et implémentée dans le compilateur ESTEREL V3. Nous donnerons seulement une idée de l'exécution de `Emit` et `Test` (de presence de signaux) dans le cas de signaux purs. La fonction $\texttt{Executer}_P$ prend et rend un argument supplémentaire : un ensemble de signaux dans lequel un signal `S` présent est noté $\texttt{S}^+$ et `S` absent est noté $\texttt{S}^-$. Afin de simplifier la lecture, on remplace, dans les instructions `Test`, l'index de l'action de test directement par le nom du signal testé.

- L'émission d'un signal ne se fait que si le signal n'est pas déjà fixé présent ou absent :

  $\texttt{Executer}_P\ [\![\texttt{n} : \texttt{Emit} : [\texttt{S}]\ (\texttt{c})]\!], R, M_{instr}, M_{par}, H, S =$
  $R + (\texttt{c}), M_{par}, H, S \cup \{\texttt{S}^+\}$ si $\texttt{S}^+ \notin S$ et $\texttt{S}^- \notin S$.

- Si le signal à émettre est fixé présent ou absent, l'instruction `Emit` n'est pas exécutée. Elle reste dans R et l'exécution du pas 1.2 ne terminera jamais. Nous pourrions aussi déclencher une erreur d'exécution :

  $\texttt{Executer}_P\ [\![\texttt{n} : \texttt{Emit} : [\texttt{S}]\ (\texttt{c})]\!], R, M_{instr}, M_{par}, H, S =$
  $R + (\texttt{n}), M_{par}, H, S$ si $\texttt{S}^+ \in S$ ou $\texttt{S}^- \in S$.

- Un `Test` ajoute la continuation *vraie* si le signal est fixé présent :

  $\texttt{Executer}_P\ [\![\texttt{n} : \texttt{Test} : [\texttt{S}]\ (\texttt{ct}, \texttt{cf})]\!], R, M_{instr}, M_{par}, H, S =$
  $R + (\texttt{ct}), M_{par}, H, S$ si $\texttt{S}^+ \in S$.

- Un `Test` ajoute la continuation *fausse* si le signal est fixé absent :

  $\texttt{Executer}_P\ [\![\texttt{n} : \texttt{Test} : [\texttt{S}]\ (\texttt{ct}, \texttt{cf})]\!], R, M_{instr}, M_{par}, H, S =$
  $R + (\texttt{cf}), M_{par}, H, S$ si $\texttt{S}^- \in S$.

- Le test de présence sur un signal non fixé le fixe absent :

  $\texttt{Executer}_P\ [\![\texttt{n} : \texttt{Test} : [\texttt{S}]\ (\texttt{ct}, \texttt{cf})]\!], R, M_{instr}, M_{par}, H, S =$
  $R + (\texttt{cf}), M_{par}, H, S \cup \{\texttt{S}^-\}$ si $\texttt{S}^+ \notin S$ et $\texttt{S}^- \notin S$.

Définition de $\texttt{TrouverContinuation}_P$

Cette fonction est définie sur les **Return**, les **Watchdog** et les **Parallel** et n'est appelée que par des **Halt** ou des **Exit**.

- $\texttt{TrouverContinuation}_P\ [\![\texttt{p} : \texttt{Parallel} : (\texttt{c}_0, \ldots, \texttt{c}_\texttt{m}) < \texttt{f} >]\!], e, h, R, M, H = R', M', H$
  Posons $M'' = M[\texttt{p} \leftarrow (M[\texttt{p}].haltset \cup h, M[\texttt{p}].active - 1, max(M[\texttt{p}].niveau, e))]$.
  Nous distinguons 3 cas :

  (1) tous les fils du **Parallel** ne sont pas terminés : on a alors $M[\texttt{p}].active \neq 1$ qui est donc décrémenté de 1. L'ensemble des **Halt** h donné par $\texttt{TrouverContinuation}_P$ est ajouté à l'ensemble des **Halt** sur lequel le **Parallel** est bloqué. Le paramètre e de $\texttt{TrouverContinuation}_P$ permet de mettre à jour le niveau de sortie du **Parallel**.

  (2) le **Parallel** est terminé et est bloqué sur au moins un **Halt** et des **Exit** de niveau 0. Le **Parallel** signale donc à son père que son niveau de terminaison est 1 et lui transmet l'ensemble des **Halt** sur lequel il est bloqué (calculé comme dans le cas 1) : il rappelle $\texttt{TrouverContinuation}_P$ et c'est le seul des trois cas dans lequel il y a un appel récursif de cette fonction.

  (3) le **Parallel** est terminé et est bloqué soit sur uniquement des **Exit** de niveau 0, soit sur au moins un **Exit** de niveau supérieur ou égal à 2. Le **Parallel** sélectionne donc une continuation en fonction de son niveau de sortie : l'instruction correspondant à cette continuation est ajoutée dans le reste R des instructions à exécuter.

  Formellement :

  (1) $R', M' = R, M''$ si $M''[\texttt{p}].active \neq 0$,

  (2) $R', M' = \texttt{TrouverContinuation}_P\ \texttt{P[f]}, 1, M''[\texttt{p}].haltset, R, M''$
  si $M''[\texttt{p}].active = 0$ et $M''[\texttt{p}].niveau = 1$,

  (3) $R', M', H = R + \texttt{c}_{max(M''[\texttt{p}].niveau - 1, 0)}, M'', H$
  si $M''[\texttt{p}].active = 0$, $M''[\texttt{p}].niveau \neq 1$ et $M''[\texttt{p}].niveau \leq \texttt{m} + 1$.
  Rappelons que **m** est le nombre de continuations moins une du **Parallel** donné dans $[\![\texttt{p} : \texttt{Parallel} : (\texttt{c}_0, \ldots, \texttt{c}_\texttt{m}) < \texttt{f} >]\!]$.

- Lorsque $\texttt{TrouverContinuation}_P$ est appelée sur un **Return**, on sait que toutes les instructions **Parallel** sont terminées et n'ont pas sélectionné de continuation (voir le cas 2 du **Parallel**) : l'exécution est finie. L'ensemble des points d'arrêt calculé est conservé dans H :

  $\texttt{TrouverContinuation}_P\ [\![\texttt{n} : \texttt{Return} : \texttt{r}]\!], e, h, R, M, H = (), M, H \cup h$

- Les **Watchdog** sont de simples relais :

  $\texttt{TrouverContinuation}_P\ [\![\texttt{n} : \texttt{Watchdog} : \{\texttt{r}\} < \texttt{f} >]\!], e, h, R, M, H =$
  $\texttt{TrouverContinuation}_P\ \texttt{P[f]}, e, h, R, M, H$

Remarque pour ESTEREL : un `Parallel` ne connaît pas toujours toutes les continuations possibles en fonction des niveaux de sortie de ses fils. Dans ce cas, il transmet son niveau de sortie au premier `Parallel` qui l'englobe. Soit, avec les notations précédentes :

$R', M', H = \texttt{TrouverContinuation}_{\texttt{P}}\ \texttt{P[f]}, max(M[\texttt{p}].niveau, e) - m, \emptyset, R, M'', H$
si $M[\texttt{p}].active = 1$ et $max(M[\texttt{p}].niveau, e) > m + 1$.

Définition de $\texttt{Reconstruire}_{\texttt{P}}$

- $\texttt{Reconstruire}_{\texttt{P}}\ \{h_1, \ldots, h_n\} = M_n, Start_n$ se calcule pas à pas à l'aide de la fonction auxiliaire $\texttt{Reconst}_{\texttt{P}}$:

$$\begin{aligned} M_1, Start_1 &= \texttt{Reconst}_{\texttt{P}}\ \texttt{P}[h_1], h_1, M_\perp, () \\ \forall i, 1 < i < n, M_i, Start_i &= \texttt{Reconst}_{\texttt{P}}\ \texttt{P}[h_i], h_i, M_{i-1}, Start_{i-1} \end{aligned}$$

Définition de $\texttt{Reconst}_{\texttt{P}}$

La fonction $\texttt{Reconst}_{\texttt{P}}$ code précisément ce qu'on a dit dans les exemples d'exécution.

- L'index d'un `Halt` s'ajoute aux instructions associées de son père :

  $\texttt{Reconst}_{\texttt{P}}\ [\![\texttt{h} : \texttt{Halt} :< \texttt{f} >]\!], index, M, Start = \texttt{Reconst}_{\texttt{P}}\ \texttt{P[f]}, \texttt{h}, M, Start$

  On note que l'*index* passé en argument n'est pas utilisé.

- L'index d'un `Parallel` s'ajoute aux instructions associées de son père :

  $\texttt{Reconst}_{\texttt{P}}\ [\![\texttt{p} : \texttt{Parallel} : (\texttt{c}_0, \ldots, \texttt{c}_\texttt{m}) < \texttt{f} >]\!], index, M, Start =$
  $\texttt{Reconst}_{\texttt{P}}\ \texttt{P[f]}, \texttt{p}, M[\texttt{p} \leftarrow M[\texttt{p}] + (index)], Start$

- Le champ *garde* d'un `Watchdog` s'ajoute aux instructions associées de son père :

  $\texttt{Reconst}_{\texttt{P}}\ [\![\texttt{n} : \texttt{Watchdog} : \{\texttt{r}\} < \texttt{f} >]\!], index, M, Start =$
  $\texttt{Reconst}_{\texttt{P}}\ \texttt{P[f]}, \texttt{r}, M[\texttt{n} \leftarrow M[\texttt{n}] + (index)], Start$

- Un `Return` arrête la récursion et ajoute l'index qu'il reçoit dans l'ensemble des points d'entrée :

  $\texttt{Reconst}_{\texttt{P}}\ [\![\texttt{n} : \texttt{Return} : \texttt{r}]\!], index, M, Start = M, Start + (index)$

C.2.3 À propos de la reconstruction

À la lecture de ce qui précède, il semble que la reconstruction calcule *dynamiquement* un grand nombre de données. En fait, en appliquant $\texttt{Reconstruire}_{\texttt{P}}$ à l'ensemble des `Halt` du programme P, on obtient pour les `Watchdog` et les `Parallel` un ensemble *statique* d'instructions associées. Dans la reconstruction présentée plus haut, les ensembles d'instructions associées calculés dynamiquement seront toujours un sous-ensemble de l'ensemble statique correspondant. En fait, la reconstruction *sélectionne* dans un ensemble statique les instructions qui seront activées dans l'instant suivant.

Annexe D

Sémantique de GC

D.1 Modèle dataflow complet de GC

D.1.1 Préordres

Préordres, exécutions

Ici, nous cherchons à modéliser les objets suivants: un ensemble fini de "ports" dont chacun porte un "flot", c'est-à-dire une suite de valeurs, un comportement de cet ensemble de ports, et un (pré)ordonnancement permettant de décrire l'ordre de disponibilité qui est affecté à toutes ces valeurs.

Ports. On se donne un ensemble S au plus dénombrable d'*occurrences*; chaque occurrence de S est affectée d'une étiquette $X \in \mathbf{X}$, où $\mathbf{X}$ est un ensemble *fini* de *ports*. On notera $X(s)$ pour désigner le port de l'occurrence s, et on désignera par $S(X)$ l'ensemble des occurrences de port X.

Le temps. Pour tout port X, l'ensemble $S(X)$ est ordonné en une *suite* que nous notons $X_1, X_2, X_3, \ldots$, dont $n = 1, 2, 3, \ldots$ est "l'indice temps". L'ensemble des occurrences X_n, où X parcourt l'ensemble des ports, est appelé *n-ième événement.* [1] On appellera *horloge* une sous-suite de l'ensemble des entiers naturels. L'ensemble des entiers $\mathbf{N}$ lui-même sera appelé l'horloge "*base*". On peut donc définir sur les horloges les opérations ensemblistes usuelles

$K = G \cup H$	réunion
$K = G \cap H$	intersection
$K = G \ominus H$	complémentaire de H dans G

Les flots synchrones. Nous modélisons maintenant les valeurs portées par les ports. On associe à chaque point $s \in S$ une valeur $v(s) \in V(X)$, où X est le port associé à s, et $V(X)$ est un domaine de valeurs (le type de X), et l'on pose $V = \coprod_{X \in \mathbf{X}} V(X)$, où $\coprod$ désigne la réunion disjointe. Nous emploierons souvent la notation v_n^X pour désigner $v(X_n)$. Le *flot synchrone* (ou tout simplement *flot* dans la suite) porté par le port X est

[1] Attention, cet indice temps ne fait référence à aucune notion de temps physique, il s'agit juste d'une numérotation abstraite.

la suite des valeurs portées par les occurrences de port X[2]. Tous les domaines $V(X)$ sont supposés contenir un élément distingué, noté $\perp$, à interpréter comme le statut "absent"; inversement, si s porte une valeur différente de $\perp$, il sera dit "présent" [3]. Pour $X \in \mathbf{X}$, l'*horloge* de X, notée H_X, est la sous-suite $(n_k)_{k=1,2,...}$ numérotant les occurrences présentes du port X. Il sera commode d'identifier horloges et flots "purs", c'est-à-dire des flots dont le domaine est réduit, hormis $\perp$, à une valeur unique que nous conviendrons alors de noter $\top$, et qui symbolise la présence; l'horloge définie par un flot pur est celle de ses instants de présence.

Les (pré)ordonnancements. Nous définirons un *préordre* sur S comme étant la donnée d'une relation $\preceq$, réflexive et transitive sur l'ensemble des occurrences S. La donnée d'un graphe orienté (admettant éventuellement des circuits) dont les sommets appartiennent à S engendre un préordre unique, par fermeture réflexive et transitive. On demande que $\{S, \mathbf{X}, \preceq\}$ satisfasse aux conditions suivantes de *causalité*

$$\forall X_n, Y_m \in S \ : \ X_n \preceq Y_m \ \Rightarrow \ n \leq m \tag{D.1}$$

(les dépendances ne peuvent remonter le temps) et de *chaîne*

$$\forall X \in \mathbf{X}, \ n \leq m \ \Rightarrow \ X_n \preceq X_m \tag{D.2}$$

Par contre, on peut avoir $s \preceq s'$ et $s' \preceq s$ à condition que s et s' appartiennent au même événement.

Les exécutions. Nous appellerons *exécution* un t-uple $\{S, \mathbf{X}, \preceq, v\}$ satisfaisant aux conditions (D.1,D.2) ci-dessus introduites. Les exécutions seront génériquement désignées par la notation $\mathcal{E}$.

INTERPRÉTATION : une exécution décrit tout à la fois un comportement (suite d'événements consistant en l'assignation de valeurs à l'ensemble des ports participant à l'événement) et des directives pour l'ordre d'évaluation de ces valeurs (la relation $s \preceq s'$ s'interprète comme "s' ne peut être évalué avant s"; en particulier, $\{s \preceq s' \text{ et } s' \preceq s\}$ décrit l'évaluation de $\{s, s'\}$ comme une action atomique — non interruptible —).

Relation d'ordre $\leq_\mathcal{E}$ sur les exécutions et les ensembles d'exécutions

Nous dirons que

$$\mathcal{E}_1 = \{S_1, \mathbf{X}_1, \preceq_1, v_1\} \ \leq_\mathcal{E} \ \{S_2, \mathbf{X}_2, \preceq_2, v_2\} = \mathcal{E}_2$$

($\mathcal{E}_1$ est un préordre étiqueté "moins partiel" que $\mathcal{E}_2$) si les conditions suivantes sont satisfaites :

1. $S_1 = S_2$; $\mathbf{X}_1 = \mathbf{X}_2$; $v_1 = v_2$ [4]

[2] La notion de "port" n'apparait pas dans GC, ce rôle est tenu par les noms de flot.

[3] La notion de flot synchrone se différencie de celle de "flot" dans l'acception classique des modèles dataflow par cette valeur particulière $\perp$ dénotant l'absence, notion inconnue dans les modèles dataflow; le terme de "trace" pourrait être plus approprié pour désigner les flots synchrones, mais nous nous en tenons à la terminologie d'usage.

[4] stricto sensu, il n'est pas nécessaire que $S_1 = S_2$, dans la mesure où l'ensemble d'occurrences S est défini à une bijection près, il s'agit là d'un abus de langage.

2. $\preceq_2 \subseteq \preceq_1$, autrement dit $\preceq_1$ est un renforcement de $\preceq_2$.

Autrement dit les deux exécutions ont mêmes ensembles d'occurrences, mêmes ports, mêmes fonctions d'étiquetage, et ne diffèrent que par leurs préordres ; et le préordre du second contient *moins* de branches que celui du premier (et non pas l'inverse, attention). On vérifie aisément que $\leq_\mathcal{E}$ est un ordre.

Cette relation d'ordre sur les exécutions s'étend aux *ensembles* d'exécutions de la façon suivante : si Π_1 et Π_2 désignent deux tels ensembles, on notera

$$\Pi_1 \leq_\mathcal{E} \Pi_2 \tag{D.3}$$

si

$$\exists\, \varphi \ :\ \Pi_1 \mapsto \Pi_2\ ,\ \varphi \text{ injective, telle que, } \forall \mathcal{E}_1 \in \Pi_1,\ \mathcal{E}_1 \leq_\mathcal{E} \varphi(\mathcal{E}_1)$$

On vérifie encore que (D.3) est bien un ordre sur les ensembles d'exécutions. De plus, on vérifie aisément que toute famille bien ordonnée d'exécutions admettant un majorant admet une borne supérieure, ceci nous servira lors de la définition de la composition des programmes.

D.1.2 Programmes et leurs combinateurs

Soient $\mathcal{E}$ et $\mathcal{E}'$ deux exécutions. Nous dirons que $\mathcal{E}'$ est une *dilatation* de $\mathcal{E}$, ou, de manière équivalente, que $\mathcal{E}$ est une *compression* de $\mathcal{E}'$, si $\mathcal{E}$ s'obtient en effaçant dans $\mathcal{E}'$ un ensemble fini d'événements *silencieux,* c'est-à-dire d'événements dont tous les ports sont absents, et en prenant la restriction à $\mathcal{E}$ des ordres et préordres définis sur $\mathcal{E}'$. Nous dirons que $\mathcal{E}'$ est une dilatation *stricte* de $\mathcal{E}$ si, de plus, $\mathcal{E}'$ est différent de $\mathcal{E}$. Nous dirons qu'un ensemble d'exécutions est *saturé par compression* si toute exécution admettant au moins un instant silencieux admet dans cet ensemble une compression stricte. Inversement, nous dirons qu'un ensemble d'exécutions est *saturé par dilatation* si toute exécution admet, pour tout n, une dilatation stricte obtenue par insertion, entre les n-ième et $(n+1)$-ième événements, d'événements silencieux.

Programmes

Définition 1 (programmes) *Un programme sur* $\mathbf{X}$ *est la donnée d'un ensemble d'exécutions sur* $\mathbf{X}$*, saturé par dilatation et compression. Les programmes seront génériquement désignés par* Π*. La relation d'ordre* $\leq_\mathcal{E}$ *définie sur les ensembles d'exécutions définit donc une relation d'ordre sur les programmes.*

Restriction et composition

Soit $\mathcal{E} = \{S, \mathbf{X}, \preceq, v\}$ une exécution sur $\mathbf{X}$, et soit $\mathbf{Y} \subset \mathbf{X}$. La *restriction* de $\mathcal{E}$ à $\mathbf{Y}$ sera une exécution $\mathcal{E}' = \{S', \mathbf{Y}, \preceq', v'\}$ sur $\mathbf{Y}$, où

- S' est la restriction de S aux occurrences de port $Y \in \mathbf{Y}$;
- $\preceq', v'$ sont les restrictions de $\preceq, v$ à S'.

Soit Π un programme sur $\mathbf{X}$, et soit $\mathbf{Y} \subset \mathbf{X}$. L'ensemble des restrictions à $\mathbf{Y}$ des exécutions de Π définit, après saturation par compression, un nouveau programme sur $\mathbf{Y}$, que nous appellerons la *restriction* de Π à $\mathbf{Y}$, et que nous noterons

$$\Pi_{!!\mathbf{Y}}$$

Soit maintenant, pour $i = 1, 2$, Π_i un programme sur $\mathbf{X}_i$. Nous appellerons *composition* de Π_1 et Π_2, notée

$$\Pi_1 | \Pi_2$$

le plus grand (pour l'ordre défini sur les programmes) programme sur $\mathbf{X_1} \cup \mathbf{X_2}$ dont la restriction à $\mathbf{X}_1$ (resp. $\mathbf{X}_2$) est $\leq_\mathcal{E} \Pi_1$ (resp. $\leq_\mathcal{E} \Pi_2$), où $\leq_\mathcal{E}$ est l'ordre partiel défini sur les programmes.

INTERPRÉTATION. En ce qui concerne les "comportements" (cf. plus loin), c'est-à-dire les suites légales de valeurs d'un programme, l'ordre défini sur les programmes coïncide avec l'inclusion ; en ce qui concerne les ordonnancements, il coïncide avec l'affaiblissement de ceux-ci. Par conséquent, composer deux programmes revient à :

- pour ce qui concerne les comportements, prendre les intersections des deux programmes restreints aux ports communs ;
- pour ce qui concerne les ordonnancements, imposer les ordonnancements qui résultent de l'un ou l'autre des deux programmes.

D.1.3 Spécifications comportementales, spécifications de synchronisation, schémas d'exécution

Spécifications comportementales

Soit $\mathcal{E} = \{S, \mathbf{X}, \preceq, v\}$ une exécution sur $\mathbf{X}$. Nous lui associons un *comportement* $\{S, \mathbf{X}, v\}$ obtenu simplement en supprimant la relation $\preceq$. De façon générale, nous appellerons *comportement* sur $\mathbf{X}$ un t-uple $T = \{S, \mathbf{X}, v\}$ satisfaisant aux conditions définies plus haut pour ces objets.

Définition 2 (spécification comportementale) *Une spécification comportementale (ou spécification) sur* $\mathbf{X}$ *est un ensemble de comportements sur* $\mathbf{X}$, *saturé par dilatation et compression.*

Nous noterons génériquement Σ les spécifications, et nous définissons, comme pour les programmes, la *restriction*

$$\Sigma_{!!\mathbf{Y}}$$

où $\mathbf{Y} \subset \mathbf{X}$, et la *composition* [5]

$$\Sigma_1 | \Sigma_2$$

De cette manière, à tout programme Π est associée une spécification unique, que nous noterons

$$\Sigma(\Pi) \tag{D.4}$$

et l'on a

$$\begin{aligned} \Sigma(\Pi_{!!\mathbf{Y}}) &= \Sigma(\Pi)_{!!\mathbf{Y}} \\ \Sigma(\Pi_1 | \Pi_2) &= \Sigma(\Pi_1) | \Sigma(\Pi_2) \end{aligned}$$

Autrement dit, $\Pi \mapsto \Sigma(\Pi)$ est un homomorphisme.

[5] se rappeler que, pour les spécifications, l'ordre retenu est celui de l'inclusion.

Spécifications de synchronisation

Soit $T = \{S, \mathbf{X}, v\}$ un comportement sur $\mathbf{X}$. Nous lui associons une *synchronisation* $\{S, \mathbf{X}, h\}$, qui est obtenue en composant la fonction d'étiquetage v avec l'application qui laisse $\perp$ invariant, et qui, à toute valeur présente, associe le symbole $\top$ (les valeurs sont perdues, mais l'information concernant les horloges est préservée). De façon générale, nous appellerons *synchronisation* sur $\mathbf{X}$ un t-uple $\{S, \mathbf{X}, h\}$ satisfaisant aux conditions définies plus haut pour ces objets.

Définition 3 (spécification de synchronisation) *Une spécification de synchronisation sur* $\mathbf{X}$ *est un ensemble de synchronisations sur* $\mathbf{X}$, *saturé par dilatation et compression.*

Nous noterons génériquement $\mathcal{H}$ les spécifications de synchronisation, et nous définissons, comme pour les programmes, la *restriction*

$$\mathcal{H}_{!!\mathbf{Y}}$$

où $\mathbf{Y} \subset \mathbf{X}$, et la *composition*

$$\mathcal{H}_1 | \mathcal{H}_2$$

De cette manière, à tout programme Π est associée une spécification de synchronisation unique, que nous noterons

$$\mathcal{H}(\Pi)$$

et l'on a

$$\begin{aligned} \mathcal{H}(\Pi_{!!\mathbf{Y}}) &= \mathcal{H}(\Pi)_{!!\mathbf{Y}} \\ \mathcal{H}(\Pi_1 | \Pi_2) &= \mathcal{H}(\Pi_1) | \mathcal{H}(\Pi_2) \end{aligned}$$

Autrement dit, $\Pi \mapsto \mathcal{H}(\Pi)$ est un homomorphisme.

Schémas d'exécution

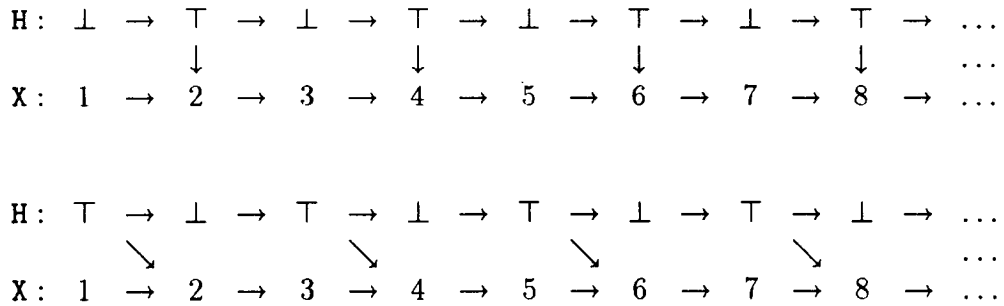
Soit $\mathcal{E} = \{S, \mathbf{X}, \preceq, v\}$ une exécution sur $\mathbf{X}$. Nous lui associons un *ordonnancement* $\mathcal{G} = \{S_{\text{pres}}, \mathbf{X}, \preceq, v\}$ obtenu simplement en supprimant les occurrences $s \in S$ affectées de la valeur $\perp$. On notera que l'on perd ainsi la notion de simultanéité, qui repose tout à la fois sur l'existence d'un temps global à l'exécution et la propriété de causalité de $\preceq$ vis-à-vis de ce temps global.

Définition 4 (schémas d'exécution) *Nous appellerons schéma d'exécution sur* $\mathbf{X}$ *un ensemble* Γ *d'ordonnancements sur* $\mathbf{X}$.

A tout programme Π est donc associé, comme indiqué plus haut, un unique schéma d'exécution, que nous noterons

$$\Gamma(\Pi)$$

NOTA : *Nous ne munissons pas les schémas d'exécution des combinateurs .!! (restriction) ni | (composition), car l'application* $\Pi \mapsto \Gamma(\Pi)$ *ne serait pas un homorphisme.* En effet, considérons les deux exécutions suivantes



où `X,H` sont les ports, et où les valeurs des occurrences ont simplement été inscrites aux emplacements de ces occurrences. Elles sont différentes, mais définissent le même préordre étiqueté après effacement des $\bot$. Par conséquent, si l'on considère les programmes Π_1 et Π_2 obtenus en saturant ces exécutions par dilatation (en acceptant par exemple tous les ordonnancements pour les événements silencieux), $\Pi_1|\Pi_2$ est le programme qui ne comporte que des événements silencieux, donc $\Gamma(\Pi_1|\Pi_2) = \emptyset$. En revanche, $\Gamma(\Pi_1) = \Gamma(\Pi_2) = \Gamma(\Pi_1)|\Gamma(\Pi_2)$ n'est pas vide [6].

INTERPRÉTATION : On notera que $\Gamma(\Pi)$ est une description en ordres partiels du programme Π, et peut donc être interprété comme une spécification d'exécution *purement asynchrone* de ce programme : toute mise en œuvre asynchrone (par file non bornée, rendez-vous, etc.) respectant cette spécification sera "correcte" en un sens que nous définirons plus loin.

D.2 Modèle pour la "partie impérative"

Le modèle de la section précédente est en toute rigueur complet. Ce que nous faisons dans cette section est simplement de fournir des moyens commodes de définir des programmes au sens de la section précédente. Le principe utilisé en est le suivant : nous considérerons maintenant des programmes enrichis d'une "partie impérative", qui nous servira pour l'essentiel à parler commodément du passé (tant en ce qui concerne les valeurs que les ordonnancements). Cette partie impérative sera purement locale au programme, et ne servira donc pas pour les opérations de restriction et de communication ; nous ne redéfinirons donc pas comment ces combinateurs agissent sur les programmes enrichis de leur partie impérative, puisqu'ils ne font qu'agir sur le programme résultant au sens de la section précédente. Voyons cela de plus près en reprenant un à un les objets introduits à la sous-section D.1.1.

D.2.1 Exécutions avec partie impérative

Variables. L'ensemble d'occurrences est élargi : il contient maintenant des occurrences s affectées d'un nouveau type d'étiquette $\xi \in \Xi$, où Ξ est un ensemble fini de *variables,* et l'on utilise les mêmes notations pour les variables que pour les ports : $\xi(s), S(\xi)$. Les

[6] Une autre raison de ne pas munir les schémas d'exécution de la composition | est que l'on peut montrer que celle-ci ne préserve pas la rationnalité.

notions relatives au temps sont étendues aux variables. L'ensemble $S(\xi)$ des occurrences de ξ est ordonné en une suite, à ceci près qu'*une même variable peut admettre un nombre supérieur à 1 (mais néanmoins fini) d'occurrences au sein d'un événement.* Par conséquent, les occurrences de variables sont notées $\xi_{1,1}, \xi_{1,2}, \xi_{1,3}, \xi_{2,1}, \xi_{2,2}, \ldots$, selon l'ordre lexicographique, où le premier index numérote l'événement, et le second la place dans l'événement.

Les (pré)ordonnancements. De même, on suppose le préordre $\preceq$ étendu à l'ensemble des occurrences, qu'elles soient associées à des ports ou à des variables ; les conditions (D.1,D.2) sont également étendues, et prennent donc, compte tenu de la numérotation lexicographique, les formes respectives

$$\forall X_{n_1,p_1}, Y_{n_2,p_2} \in S \ : \ X_{n_1,p_1} \preceq Y_{n_2,p_2} \ \Rightarrow \ n_1 \leq n_2 \tag{D.5}$$

$$\forall \xi \in \Xi \ , \ (n_1,p_1) \leq (n_2,p_2) \ \Rightarrow \ \xi_{n_1,p_1} \preceq \xi_{n_2,p_2} \tag{D.6}$$

Dans la première condition, les occurrences sont de tout type (port ou variable), et la numérotation standard en n pour les ports est identifiée à la double numérotation $(n,1)$ où la composante supplémentaire est à 1. On notera que (D.5) signifie que le temps ne peut être remonté, la seconde composante, par contre, peut l'être.

Valeurs. La fonction de valuation v est étendue aux occurrences de variables, et nous noterons en particulier $v^{\xi}_{1,1}, v^{\xi}_{1,2}, v^{\xi}_{1,3}, v^{\xi}_{2,1}, \ldots$ pour désigner $v(\xi_{1,1}), v(\xi_{1,2}), v(\xi_{1,3}), v(\xi_{2,1}), \ldots$, qui sont les valeurs des occurrences successives de ξ. Comme précédemment, si s est une occurrence de ξ, $v(s) = \perp$ signifie que la variable ξ n'est pas accessible à l'occurrence considérée. L'ensemble des instants de présence de ξ est l'horloge de ξ et est noté H_ξ. Toute fonction de valuation v associée à une variable jouit de la propriété suivante (qui justifie le nom de "variable") : *il existe une sous-suite* $S_{\text{write}}(\xi) \subset S(\xi)$, *telle que*

$$\begin{aligned} \forall s \in S_{\text{write}}(\xi) \ &\Rightarrow \ v(s) \neq \perp \\ \forall s \in S(\xi) \ , \ v(s) \neq \perp \ &\Rightarrow \ v(s) = v\left(\max_{s' \in S_{\text{write}}(\xi)} s'\right) \end{aligned} \tag{D.7}$$

où le max fait référence au préordre $\preceq$ sur l'exécution considérée. Bien entendu, nous dirons que $S_{\text{write}}(\xi)$ constitue l'ensemble des *occurrences d'écriture* de la variable ξ. La condition (D.7) signifie alors que, entre ses occurrences d'écriture, la variable ξ est, ou bien absente, ou bien non modifiée. Rien n'empêche qu'un événement comporte plusieurs occurrences d'écriture d'une même variable.

Exécution induite. Nous disposons maintenant d'*exécutions avec partie impérative*, qui sont des t-uples de la forme $\{S, \mathbf{X} \cup \Xi, \preceq, v\}$, où le nouvel objet introduit est l'ensemble Ξ des variables. Les exécutions avec partie impérative seront génériquement notées $\mathcal{I}$. Bien entendu, à toute $\mathcal{I}$ nous pouvons associer une exécution $\mathcal{E}$ en effaçant simplement les occurrences des variables et en prenant la restriction du préordre. A partir d'un ensemble de $\mathcal{I}$, on obtient ainsi un ensemble de $\mathcal{E}$.

D.2.2 Programmes avec partie impérative

Définition 5 (programmes avec partie impérative) *Un programme avec partie impérative sur* $\mathbf{X} \cup \Xi$ *est la donnée d'un ensemble d'exécutions avec partie impérative sur*

```
                 metagc Prog_Name
                 act List_Act_Name
prog    ::=      var List_Var_Name
                 Body
                 end Prog_Name

Body    ::=  Body | Body | Clockeq | Preceq | Floweq | Imper | ( Body )

Clockeq ::=  clock ( Name ) = Clock | Clock = Clockexp
Clockexp::=  Clock Clockop Clock | tt ( bool )
Clockop ::=  default | when | whennot

Preceq  ::=  Name --> Name at Clock
Name    ::=  Flow_Name | Act_Name

Floweq  ::=  Flow_Name = Flowexp at Clock

Imper   ::=  Act_Name : Act
Act     ::=  input | output
input   ::=  Var_Name = Flow_Name at Clock
output  ::=  Flow_Name = Varexp at Clock
```

Tableau D.1 : *Le langage "amande"* μGC

$\mathbf{X} \cup \Xi$, *qui engendre, par effacement des actions et variables, un ensemble d'exécutions sur* $\mathbf{X}$, *saturé par dilatation et compression.*

On notera que nous n'étendons pas aux programmes avec partie impérative l'ordre partiel $\leq_\varepsilon$, ni les combinateurs "restriction" et "composition". En effet, les parties impératives *ne jouent aucun rôle dans ces opérations,* que l'on applique exclusivement sur les programmes induits par effacement des variables. En fait, comme on le verra, les variables ne sont là que pour faciliter l'expression de dépendances et de relations entre flots portant sur des instants différents.

D.3 Utilisation du modèle pour la sémantique de GC

D.3.1 μGC

La table D.1 décrit le métalangage [7] suivant, obéissant au doux nom de μGC, qui nous servira ensuite à coder directement GC. Les symboles-clé du langage sont encadrés. Outre les symboles-clé, les terminaux de ce langage sont, dans l'ordre d'entrée en scène, "Clock" (une horloge), "bool" (un flot booléen), "Flow_Name" (nom de flot), "Act_Name" (nom d'action), "Var_Name" (nom de variable), "Flowexp" (expression de flot), "Var"

[7] En ce qui concerne le titre, on rappelle qu'une amande est le noyau d'un noyau.

(variable), "Varexp" (expression de variables). Le langage est très informellement décrit, dans la mesure où les terminaux sont un peu macroscopiques. Par exemple, le petit programme μGC suivant...

```
metagc PRE                                  -- nom du programme

  act STATE, OUTPUT                         -- declaration des actions locales
  var Xi                                    -- declaration des variables locales

    ( ( clock(IN)     = H                   -- equations d'horloge
      | clock(OUT)    = H
      | clock(STATE)  = H
      | clock(OUTPUT) = H )
    | ( OUTPUT --> OUT   at H               -- dependances
      | OUTPUT --> STATE at H
      | IN     --> STATE at H
      | H      --> IN    at base
      | H      --> OUT   at base )
    | ( STATE  : Xi = IN at H               -- actions
      | OUTPUT : OUT = Xi at H ) )

end PRE
```

... équivaut au `pre` de GC. Afin de préciser ceci, nous donnons la sémantique des équations de μGC (les déclarations sont du sucre syntaxique que nous laissons de côté).

La première question que nous examinons est la sémantique des actions, dans la mesure où cette notion n'existe pas dans notre modèle abstrait. Considérons un programme μGC. Regroupons toutes les actions partageant une même variable ξ (par exemple, les actions `STATE` et `OUTPUT` partagent la variable `Xi` dans le programme `PRE`). Ces actions définissent alors des sous-ensembles de l'ensemble des occurrences de la variable ξ, et, pour des actions distinctes, ces sous-ensembles sont distincts. Les équations d'horloge `clock(STATE) = ...` etc., spécifient les instants de présence de l'action référencée, tandis que l'horloge de présence des occurrences de la variable ξ est la réunion des horloges des actions partageant cette variable. Bien entendu, lorsque deux telles actions partagent un même instant de présence, cela crée ipso-facto *deux* occurrences de la variable ξ à cet instant, avec les deux entrelacements possibles (action (a) avant action (b) ou l'inverse). Le recours aux ordonnancements sur les actions permet, si désiré, de fixer la numérotation voulue. Ainsi, dans le programme `PRE` , la variable `Xi` est partagée par les deux actions `STATE` et `OUTPUT` , qui ont même horloge `H` . La variable `Xi` admet donc deux occurrences de présence, à chaque instant de présence de `H` . La dépendance " `OUTPUT --> STATE at H` " ordonne ces actions à chaque instant de `H` . Dans le cas présent, elle spécifie que l'émission de la valeur courante de ξ (action `OUTPUT`) doit se faire avant l'écriture dans ξ de la nouvelle valeur de `IN` (action `STATE`).

Vérifions que nous obtenons bien le `pre` de GC. A cette fin, il sera commode d'introduire l'indice n numérotant les événements de présence de l'horloge `H` , horloge unique de tous les flots ou actions du programme. Le programme `PRE` ci-dessus se traduit ainsi, en notant x pour `IN` et y pour `OUT` :

$$\forall n > 0 \ : \ x_{n-1} \rightarrow \xi_{n-1,2} \rightarrow \xi_{n,1} \rightarrow y_n$$

$$\forall n > 0 \ : \ y_n = \xi_{n,1} = \xi_{n-1,2} = x_{n-1}$$

Si l'on efface la variable ξ, il vient exactement

$$\forall n > 0 \ : \ x_{n-1} \rightarrow y_n$$
$$\forall n > 0 \ : \ y_n = x_{n-1}$$

qui est exactement la définition de `pre` en GC. Nous allons formaliser tout ceci.

Déclaration des actions et variables.

- La déclaration

```
metagc PROG

  act ACT_1,...,ACT_p;...
  var Xi
```

où `ACT_1,...,ACT_p` est l'ensemble des actions partageant la variable `Xi`, crée une variable ξ possédant p occurrences à chaque événement. Nous désignerons par $\xi_{n,\texttt{ACT\_1}}, \ldots, \xi_{n,\texttt{ACT\_p}}$ ces occurrences au n-ième événement. Attention, l'ordre de ces occurrences n'est pas spécifié (la numérotation des actions n'est ici qu'une notation de commodité), sauf ordonnancement explicite à l'aide de l'instruction `-->` introduite ci-après.

Équations d'horloge. Cette partie de la sémantique correspond aux définitions

Clockeq	::=	`clock` `(` Name `)` `=` Clock | Clock `=` Clockexp
Clockexp	::=	Clock Clockop Clock | `tt` `(` bool `)`
Clockop	::=	`default` | `when` | `whennot`

- `H = K default/when/whennot L` correspond aux équations d'horloge $H = K \cup / \cap / \ominus L$
- `clock(X) = H` correspond à $H_X = H$
- `H = tt(B)` désigne l'ensemble des exécutions sur les ports `(H,B)`, où `H` est de type pur et `B` est booléen, telles que $v_n^H = \top$ si et seulement si $v_n^B = tt$.

Dépendances. Nous donnons la sémantique de

Preceq	::=	Name `-->` Name `at` Clock
Name	::=	Flow_Name | Act_Name

- `X --> Y at H` est ici défini dans le cas où, par exemple, `X` est une action, et `Y` un port. Soient $\xi^1, \ldots, \xi^p$ les variables invoquées dans l'action `X`. Alors, cette instruction crée, pour tout instant n de présence de l'horloge `H`, et pour $i = 1, \ldots, p$, les précédences $\xi^i_{n,\texttt{X}} \preceq Y_n$, où $\xi^i_{n,\texttt{X}}$ est l'occurrence de la variable ξ associée à l'action `X` à l'événement n considéré. Les autres cas sont définis similairement.
- `(X_1,...,X_p) --> (Y_1,...,Y_q) at H` est une notation abrégée pour `X_i --> Y_j at H` pour tout couple d'indices `(i,j)`.

Équations de flot. Ceci correspond à la partie suivante de μGC :

Floweq ::= Flow_Name [=] Flowexp [at] Clock

- `X = f(Y,Z) at H` est l'ensemble des exécutions sur `(X,Y,Z,H)`, où `H` est une horloge et `X,Y,Z` sont des ports de type approprié, satisfaisant à la condition d'horloge $H \subseteq (H_X \cap H_Y \cap H_Z)$ et à la condition $v_n^X = f(v_n^Y, v_n^Z)$ pour tout instant n de présence de l'horloge H.

Partie impérative. Ceci correspond à la partie suivante de μGC :

Imper ::= Act_Name [:] Act
Act ::= input | output
input ::= Var_Name [=] Flow_Name [at] Clock
output ::= Flow_Name [=] Varexp [at] Clock

Nous avons déjà étudié la déclaration des actions. Il nous reste donc à détailler ce qu'est "Act".

- `IN_ACT : Xi = Z at H` crée, à tout événement n de présence de `H`, et pour l'occurrence $\xi_{n,\texttt{IN\_ACT}}$ de la variable ξ associée à l'action `IN_ACT`, l'égalité $v^{\xi}_{n,\texttt{IN\_ACT}} = v_n^Z$. En conséquence, `H` doit être absente quand `Z` l'est.
- `OUT_ACT : Y = f(Xi_1,...,Xi_p) at H` crée, à tout événement n de présence de `H`, et pour les occurrences respectives $\xi^1_{n,\texttt{OUT\_ACT}}, \ldots, \xi^p_{n,\texttt{OUT\_ACT}}$ des variables $\xi^1, \ldots, \xi^p$ qui sont partagées par l'action considérée,
  la contrainte $v_n^Y = f(v^{\xi^1}_{n,\texttt{OUT\_ACT}}, \ldots, v^{\xi^p}_{n,\texttt{OUT\_ACT}})$. En conséquence, `H` doit être absente quand `Y` l'est.

Les combinateurs agissant sur les programmes avec partie impérative. Leur sémantique est exactement celle des combinateurs agissant sur les programmes sans partie impérative qu'ils induisent.

REMARQUE. On aura noté que, dans le cas où deux actions différentes `ACT` et `ACT'` partagent une même variable `Xi`, l'ordre des occurrences $\xi_{n,\texttt{ACT}}$ et $\xi_{n,\texttt{ACT'}}$ est *indifférent* (à moins qu'il ne soit explicitement spécifié, comme dans l'exemple `PRE`). En particulier, dans le cas où `ACT` est une action de lecture de la variable et `ACT'` une action d'écriture dans cette variable, la non spécification de l'ordre entre ces deux actions résulte en un non-déterminisme, puisque les deux ordres sont considérés comme légaux, avec, bien entendu, des résultats différents.

D.3.2 GC en μGC

Sémantique comportementale de GC

Nous donnons, pour chaque programme GC, le codage en μGC de la spécification qui lui est associée. Autrement dit, nous ne mentionnons pas les ordonnancements, sauf lorsqu'ils sont nécessaires à la bonne définition des comportements (ordonnancement des actions). Pour chaque contexte d'exécution, des ordonnancements supplémentaires seront à ajouter, nous verrons cela sur un exemple.

Les primitives.

`clock(X) = H` se conserve en μGC

`X = f(A,B)` correspond aux équations μGC

```
( ( clock(X) = clock(A) = clock(B) = H )  -- horloges
| ( X = f(A,B) at H ) )                   -- flots
```

`Y = pre(X)` correspond au programme μGC (que nous écrivons complètement ici)

```
metagc PRE                              -- nom du programme

  act STATE, OUTPUT                     -- declaration des actions locales
  var Xi                                -- declaration des variables locales

  ( ( clock(X)     = clock(Y)      = H     -- horloges
    | clock(STATE) = clock(OUTPUT) = H )
  | ( OUTPUT --> STATE at H )              -- dependances
  | ( STATE  :  Xi = X at H                -- actions
    | OUTPUT :  Y  = Xi at H ) )

end PRE
```

`H = tt(B)` correspond aux équations μGC

```
( ( clock(B) = K                -- horloges
  | H = tt(B) ) )
```

`Y = X when H` correspond aux équations μGC

```
( ( clock(X) = CX                       -- horloges
  | clock(Y) = CY
  | CY = CX when H )
| ( Y = X at CY ) )                     -- flots
```

`Y = U default V` correspond aux équations μGC

```
( ( clock(U) = CU                       -- horloges
  | clock(V) = CV
  | clock(Y) = CY
  | CY = CU default CV
  | seulCV = CV whennot CU )
| ( Y = U at CU                         -- flots
  | Y = V at seulCV ) )
```

Équation	Dépendance induite
	`clock(X) --> X at base`
`H = K clockop L`	`(K,L) --> H at base`
`X --> Y at H`	`H --> Y at base`
`H = tt(B)` `K = clock(B)`	`B --> H at K`
`Y = f(A,B) at H`	`(A,B) --> Y at H`
`IN-ACT: Xi = X at H`	`X --> IN-ACT at H`
`OUT-ACT: X = exp at H`	`OUT-ACT --> X at H`

Tableau D.2 : *Dépendances induites*

Les combinateurs. Si μP désigne génériquement le codage en μGC de la sémantique comportementale d'un programme GC `P`, on a simplement

$$\begin{aligned} \mu(\texttt{P!!X}) &= (\mu\texttt{P})\texttt{!!X} \\ \mu(\texttt{P | Q}) &= (\mu\texttt{P})\texttt{|}(\mu\texttt{Q}) \end{aligned}$$

qui signifie que les combinateurs sont exactement les mêmes en μGC et GC.

Les dépendances et la sémantique opérationnelle

Deux sortes de dépendances sont à prendre en compte : celles qui résultent d'une dépendance explicitée dans le programme GC, et celles qui sont requises pour la sémantique opérationnelle d'un programme GC, et qui servent à expliciter les ordonnancements des évaluations. Pour la première, on a :

`X -H-> Y` correspond aux équations μGC :

```
( ( clock(X) = CX                          -- horloges
  | clock(Y) = CY
  | K = CX when CY when H )
| ( X --> Y at K ) )                       -- dependances
```

Pour le second aspect, aux équations d'horloge, de flot, et aux actions nous convenons d'associer des dépendances, dites "induites", selon les règles de la table D.2. Ces dépendances induites codent les ordonnancements adéquats pour chacune des équations mentionnées. On notera en particulier que tout flot est précédé par son horloge, et que l'indication de précédence de `X` sur `Y` à l'horloge `H` implique une précédence de

H sur Y à l'horloge base , seule horloge qui, dans tout contexte, permet de parler de la présence ou de l'absence d'une horloge. Pour obtenir la sémantique opérationnelle d'un programme GC, on procédera comme suit :

1. on code les comportements du programme GC considéré en μGC, en utilisant le codage des primitives donné plus haut, les dépendances explicitées dans le programme GC source doivent être également transcodées,

2. on peut récrire le programme μGC ainsi obtenu en un autre équivalent (définissant les mêmes comportements), les dépendances explicitées dans le programme GC source devant être inchangées,

3. on applique les règles du tableau D.2 jusqu'à saturation pour obtenir les ordonnancements nécessaires à la bonne exécution du programme.

Nous donnerons plus loin un critère permettant de savoir quand cette procédure permet effectivement d'exécuter le programme (on a alors bien construit une sémantique opérationnelle du programme GC considéré).

D.3.3 Un exemple

Voici le petit programme GC que nous allons étudier :

```
( X = U default (pre(X)-1)
| clock(U) = tt(pre(X) =< 0) )   --  =< : inferieur ou egal
```

Un comportement de ce programme est, par exemple :

U :	3	$\perp$	$\perp$	$\perp$	0	2	$\perp$	$\perp$	...
X :	3	2	1	0	0	2	1	0	...
pre(X) :	0	3	2	1	0	0	2	1	...
pre(X) $\leq$ 0 :	t	f	f	f	t	t	f	f	...
clock(U) :	$\top$	$\perp$	$\perp$	$\perp$	$\top$	$\top$	$\perp$	$\perp$	...

Autrement dit, ce programme est un "multiplexeur" ou "suréchantillonneur" : sur lecture de l'entrée U (de type entier positif ou nul), le programme délivre un nombre de "ticks" supplémentaires égal à la valeur lue pour U. Nous allons maintenant construire la sémantique de ce programme. Nous commençons par la sémantique comportementale. Appelons MUX_1 le programme μGC correspondant à la première équation, nous le décomposons en

```
metagc MUX_1                          -- nom du programme

DEFAULT_GC | PRE_GC                   -- corps
```

correspondant respectivement aux équations GC `X = U default (Y-1)` et `Y = pre(X)` . Rassemblons les codages μGC des comportements de ces deux équations GC, il vient le résultat montré à la figure D.1, 149. Nous pouvons simplifier les équations d'horloge, ce qui donne le résultat de la figure D.2, 149. On notera que cette simplification porte sur les champs "horloges", et que, pour les effectuer, *nous avons utilisé les propriétés du calcul sur l'algèbre des horloges.* Bien entendu, nous n'en avons pas terminé, puisqu'il faut maintenant prendre en compte la seconde équation.

Néanmoins, il est intéressant de considérer ce programme μGC tout seul. Tel qu'il est, il lui manque, pour être exécuté, les ordonnancements des calculs de valeurs, d'horloge, et entre flots et actions. Les ordonnancements à rajouter s'obtiennent par application jusqu'à saturation des règles de la table D.2 donnant les dépendances induites. L'ordonnancement complet pour ce programme est montré à la figure D.3, 150 (il en constitue la sémantique opérationnelle). En examinant ce programme privé de sa première équation, on constate que n'y apparaissent jamais à gauche d'une équation de définition :

- les horloges `H,CU` ,
- les valeurs du flot `U` .

Ces objets sont donc les "entrées" du programme privé de sa première équation. Cette première équation, isolée du reste du corps du programme, spécifie la contrainte à laquelle les horloges `H,CU` sont assujetties : nous n'y avons pas adjoint ses dépendances induites, puisqu'un cycle en aurait immédiatement résulté. Se référant aux divers champs des programmes GC, la première équation constituerait une *synchronisation* (auquel cas aucune dépendance induite n'est engendrée), le reste du programme étant dans le champ *définition* (où les dépendances induites sont prises en considération). On a donc ici un programme déterministe dont les entrées sont les horloges `H,CU` et les valeurs du flot `U` , mais avec des contraintes sur les horloges d'entrée `H,CU` .

Continuons l'analyse, et prenons la seconde équation GC. Son codage μGC est, en notant `B` le booléen `(Y =< 0)` , montré à la figure D.4, 150. La combinaison de `MUX_1` tel qu'à la figure D.2 et de `MUX_2` de la figure D.4 donne finalement, en supprimant les équations en double, le programme de la figure D.5, 151. Il manque au programme résultant l'ordonnancement du calcul d'horloge, nous procédons comme pour le programme `MUX_1` . Le résultat est montré à la figure D.6, 152. On constate aisément que ce programme μGC

1. est sans cycle si l'on examine les `-->` ,
2. est sans double définition de flot,
3. admet pour entrées l'horloge d'activation `H` et les valeurs du flot `U` ,
4. tous les autres objets étant calculables à partir de ces entrées, avec l'ordonnancement décrit par les `-->` en tenant compte de leurs horloges.

Autrement dit, nous avons là un programme "exécutable" et déterministe ayant pour entrées l'horloge d'activation `H` et les valeurs du flot `U` .

Mais, nous pouvons faire plus : nous pouvons exhiber, sur ce programme μGC, ce qu'est le schéma d'exécution de `MUX` , au sens de la définition 4. Il nous suffit d'effacer les $\perp$. Ceci se fait de la manière suivante :

1. on calcule la fermeture transitive du graphe `-->` ,
2. on élimine l'usage des opérateurs " `clockop` " sur horloge, qui ont réellement besoin d'un contexte où l'on puisse parler de présence et d'absence; ceci se fait en ayant seulement recours à l'opérateur `tt(.)` ; dans notre cas, on réalisera le `whennot` en produisant le booléen `C = not B` , et en remarquant que l'on a `seulH = tt(C)` ,

3. en raréfiant les horloges des dépendances pour les ramener aux occurrences présentes de leur origine.

Voici le résultat sur `MUX` : c'est le programme `MUX_ASYNCH` de la figure D.7, 153. On vérifie alors encore que ce programme μGC

1. est sans cycle si l'on examine les `-->` ,
2. est sans double définition de flot,
3. admet pour entrées l'horloge d'activation `H` et les valeurs du flot `U` ,
4. tous les autres objets étant calculables à partir de ces entrées, avec l'ordonnancement décrit par les `-->` en tenant compte de leurs horloges.

Autrement dit, nous avons là un programme "exécutable en mode asynchrone". Bien entendu, la raréfaction des horloges peut se faire à la carte.

Conclusions à propos de cet exemple. On notera les points suivants, que nous systématiserons à la section suivante :

- Nous avons utilisé du calcul formel sur les horloges pour pouvoir modifier et simplifier la forme syntaxique des programmes μGC sans en modifier la sémantique comportementale.
- Les dépendances nécessaires au bon ordonnancement du programme final recherché (`MUX`) n'ont été rajoutées qu'en phase finale, lorsqu'on dispose du programme μGC global, après les réécritures. En particulier, on notera que le programme `MUX` de la figure D.6 oriente les dépendances de `X` vers l'horloge `K` de `U` , alors que l'instruction GC `default` , considérée seule, induirait, par application des règles du tableau D.2, des dépendances en sens inverse, à savoir de `U` vers `X` . On constate bien que le choix des dépendances dépend du contexte.
- Nous avons ensuite utilisé un mécanisme de "raréfaction des horloges" pour obtenir une désynchronisation, totale ou partielle.
- Nous avons implicitement utilisé une notion de "programme exécutable".

Nous allons systématiser tout ceci à la sous-section suivante.

D.3.4 Exécutabilité, dépendances induites, désynchronisation

Exécutabilité

Nous dirons qu'un programme μGC est *correctement ordonnancé* si toute équation de ce programme a sa dépendance induite explicitée dans ce programme. Dans l'énoncé qui suit, nous appelons " `empty_clock` " l'horloge qui est toujours absente.

Définition 6 (programme μGC exécutable) *Un programme μGC, noté* `P` *, sera dit exécutable s'il satisfait aux conditions suivantes :*

1. `P` *est correctement ordonnancé ;*

2. *pour tout cycle de dépendance apparaissant dans* P *, de la forme*

```
( X_0      --> X_1 at H_1
| X_1      --> X_2 at H_2
| .......
| X_(p-1) --> X_0 at H_0 )
```

on a

```
H_1 when H_2 when ... when H_0  =  empty_clock
```

Autrement dit, le programme P *n'admet pas de circuit ;*

3. *pour toute double définition de flot apparaissant dans* P *, de la forme*

```
( X = exp_1 at H_1
| X = exp_2 at H_2 )
```

on a

```
H_1 when H_2 =  empty_clock
```

Autrement dit, la partie flot-de-données de P *satisfait à l'assignation unique ;*

4. *on considère les actions respectives d'écriture et de lecture partageant une variable particulière ; alors, à tout événement, chaque occurrence présente d'une telle action de lecture possède, parmi les actions d'écriture considérées, un plus proche prédécesseur spécifié de manière unique. Autrement dit, la partie impérative de* P *satisfait à l'assignation unique.*

On a alors le résultat suivant :

Théorème 1 (propriétés des programmes exécutables) *La restriction d'un programme exécutable à sa partie "dépendances" définit un ordre partiel. Pour tout programme exécutable de* μGC*, il existe une partition de ses ports en entrées et autres ports, et, pour cette partition, ce programme est une fonction, dont l'ordre partiel ci-dessus spécifie la sémantique opérationnelle.*

Le théorème constitue en quelque sorte une justification du label "exécutable" que nous avons conféré aux programmes satisfaisant aux conditions de la définition 6. Il exprime que cette définition fournit une condition suffisante effective pour qu'un programme μGC admette une sémantique opérationnelle. On notera que les conditions énoncées dans cette définition sont non décidables en général, à cause de l'opérateur " `tt(B)` " transformant un booléen en une horloge, le booléen `B` pouvant lui-même résulter de l'évaluation d'un prédicat sur des flots de type quelconque. Par conséquent, on implémente en pratique des abstractions de cette condition, dont voici les deux plus utiles :

- On renforce les conditions 2 et 3 de la définition 6 comme suit : il n'existe pas du tout de cycle de dépendance ni de définition multiple ; nous dirons alors que le programme μGC est *fortement* acyclique et à assignation unique.

- On conserve les conditions 2 et 3 de la définition 6, mais la vérification des conditions d'horloge vide se fait en considérant que tout flot booléen `B` résultant de l'évaluation d'un prédicat sur des types non-booléens est une entrée (c'est-à-dire que les deux valeurs *vrai* et *faux* sont considérées comme possibles dès que `B` est présent).

Le premier point de vue est celui qui est suivi par les compilateurs existants des langages LUSTRE et SIGNAL, tandis que le second, plus fin, est celui qui est suivi pour la sémantique opérationnelle de SIGNAL.

Démonstration. Le premier point est immédiat, voyons le reste de l'énoncé. Parmi les horloges manipulées, nous disposerons des deux constantes " `base` " (représentant $\top$) et " `empty_clock` " (représentant $\bot$) . Nous utiliserons les notations suivantes :

`eq ∈ P`	`eq` est une équation de `P`
`P / eq`	suppression de `eq` dans `P`
`P [X:x]`	remplacement de `X` par la valeur `x` dans `P`
`X source in P`	`X` ne figure jamais à droite du symbole " `-->` "

Nous introduisons alors les trois règles suivantes :

(R1) $$\texttt{P\_f} = \begin{array}{l}\texttt{(X --> Y at base}\\ \texttt{| Y = f(X) at base) }\end{array} \in \texttt{P} \quad \vdash \quad \texttt{(P / P\_f) [Y:f(X)]} \qquad \texttt{X source in P}$$

(R2) $$\texttt{P\_ACT} = \begin{array}{l}\texttt{(X --> ACT at base}\\ \texttt{| ACT : Xi = exp at base)}\end{array} \in \texttt{P} \quad \vdash \quad \texttt{(P / P\_ACT) [Xi:exp]} \qquad \texttt{X source in P}$$

(R3) $$\texttt{(... at empty\_clock)} \in \texttt{P} \quad \vdash \quad \texttt{P / (... at empty\_clock)}$$

Les deux premières règles expriment la condition d'évaluation des flots ou variables (`X` est une notation abrégée dénotant le t-uple `(X_1,...,X_p)` qui est l'ensemble des prédécesseurs de `Y` dans `P`), et `Y` s'évalue à la valeur `y` , égale à l'une des constantes `base` ou `empty_clock` lorsque `Y` est une horloge. La troisième règle, où "..." désigne toute expression de μGC pouvant être utilisée devant le terme " `at H` ", exprime que l'on peut effacer toute opération qui n'a pas d'occurrence présente. Ces règles codent les transitions d'un automate ; la donnée d'un programme `P` en définit l'état initial. Ainsi l'on associe à tout programme un automate, qui code les ordonnancements pour l'évaluation de ce programme. Nous allons montrer que les conditions 1 et 2 de la définition 6 impliquent que cet automate n'admet pas d'autre état puits que l'état désiré, à savoir le programme vide. Ceci montrera le théorème. Nous traitons d'abord le cas des primitives μGC, puis celui des programmes.

Les primitives. Nous traitons en détail deux cas représentatifs, à savoir les primitives suivantes :

```
X --> Y at H
Y = f(A,B) at H
```

Voyons la première. L'application des règles du tableau D.2 donne, pour `X --> Y at H` , l'ensemble {primitive + dépendances induites} suivant :

```
( X --> Y at H
| H --> Y at base )
```

Par application des règles de dérivation données plus haut, il vient :

$$\texttt{(X --> Y at H | H --> Y at base)} \xrightarrow[\text{(R1) , H=base}]{} \texttt{X --> Y at base}$$

$$\texttt{(X --> Y at H | H --> Y at base)} \xrightarrow[\text{(R3) , H=empty\_clock}]{} \texttt{nil}$$

Dans le cas où `H` s'évalue à `base` , l'automate se met en attente de l'évaluation de `X` ; dans l'autre cas, la condition d'ordonnancement disparaît. Passons à la seconde primitive. L'application des règles du tableau D.2 donne, pour `Y = f(A,B) at H` , l'ensemble {primitive + dépendances induites} suivant :

```
( Y = f(A,B) at H
| clock(A) --> A at base
| clock(B) --> B at base
| clock(Y) --> Y at base
| (A,B) --> Y at H
| H --> Y at base )
```

Le début de la dérivation donne

$$\texttt{(Y = f(A,B) at H | clock(A) --> A at base | clock(B) --> B at base | clock(Y) --> Y at base | (A,B) --> Y at H | H --> Y at base)} \xrightarrow[\text{(R1) , H=base}]{} \texttt{(Y = f(A,B) at base | (A,B) --> Y at base)}$$

$$\texttt{(Y = f(A,B) at H | clock(A) --> A at base | clock(B) --> B at base | clock(Y) --> Y at base | (A,B) --> Y at H | H --> Y at base)} \xrightarrow[\text{(R3) , H=empty\_clock}]{} \texttt{(clock(A) --> A at base | clock(B) --> B at base | clock(Y) --> Y at base)}$$

Le premier cas se réduit au programme pour l'exécution de `Y = f(A,B) at H` dans le cas monohorloge, tandis que le second se réduit au programme acceptant tous les comportements pour les ports `A,B,Y` . La dérivation finale pour le premier cas donne alors

$$\texttt{(Y = f(A,B) at base | (A,B) --> Y at base)} \xrightarrow{(R1)} \texttt{[Y:f(A,B)]}$$

Les autres primitives se traitent de la même manière.

Les programmes. Grâce à la condition 1 de la définition 6, toutes les primitives du programme μGC considéré sont équipées de leurs dépendances induites : l'exécution d'une de ces primitives est donc possible dès que les nœuds sources de l'ensemble {primitive + dépendances induites} sont évalués. La condition 2 de la définition 6 garantissent alors qu'aucun blocage n'apparaît dans l'enchaînement de ces évaluations. Les conditions 3 et 4 garantissent enfin que ces évaluations sont définies sans ambiguïté et fournissent un résultat unique. Ceci achève la démonstration du théorème. □

REMARQUE. En fait, toute cette sous-section aurait pu être écrite en remplaçant le mot " `base` " par le mot " `tick` ", où `tick` désigne la borne supérieure de toutes les horloges du programme considéré (contrairement à `base` , `tick` est donc une notion locale).

Désynchronisation

Reportons-nous au tableau D.2. La seule partie de ce tableau qui nécessite explicitement les symboles $\perp$ d'absence est l'ensemble de ses trois premières lignes. En revanche, les autres lignes ne nécessitent pas l'usage du symbole $\perp$. Un programme μGC qui ne fait apparaître aucun symbole $\perp$ est un schéma d'exécution au sens de la sous-section D.1.3, c'est-à-dire un programme exécutable de manière désynchronisée. Autrement dit, pour désynchroniser une exécution d'un programme μGC, on cherchera a récrire autant que possible les équations d'horloges du programme en ne faisant appel qu'à des équations utilisant l'opérateur `tt` . C'est exactement ce que nous avons fait pour l'exemple `MUX` . Bien entendu, des désynchronisations partielles sont également possibles.

```
metagc MUX_1

  act STATE, OUTPUT
  var Xi

    ( ( clock(U) = CU                              -- horloges
      | clock(Y) = CY
      | clock(X) = CX
      | CX = CU default CY
      | seulCY = CY whennot CU
      | clock(X)      = clock(Y)       = H
      | clock(STATE) = clock(OUTPUT) = H )
    | ( OUTPUT --> STATE at H )                    -- dependances
    | ( STATE  :  Xi = X at H                      -- actions
      | OUTPUT :  Y  = Xi at H ) )
    | ( X = U at CU                                -- flots
      | X = (Y-1) at seulCY ) )

end MUX_1
```

Figure D.1 : *Sémantique comportementale du programme* MUX_1

```
metagc MUX_1

  act STATE, OUTPUT
  var Xi

    ( ( clock(U) = CU                              -- horloges
      | H = CU default H
      | seulH = H whennot CU
      | clock(X)      = clock(Y)       = H
      | clock(STATE) = clock(OUTPUT) = H )
    | ( OUTPUT --> STATE at H )                    -- dependances
    | ( STATE  :  Xi = X at H                      -- actions
      | OUTPUT :  Y  = Xi at H ) )
    | ( X = U at CU                                -- flots
      | X = (Y-1) at seulH ) )

end MUX_1
```

Figure D.2 : *Sémantique comportementale du programme* MUX_1 *simplifiée*

```
metagc MUX_1

  act STATE, OUTPUT
  var Xi

  ( ( ( H = CU default H ) )      <== contrainte    -- horloges

  | ( ( seulH = H whennot CU                        -- horloges
      | clock(U)      = CU
      | clock(X)      = clock(Y)       = H
      | clock(STATE) = clock(OUTPUT) = H )
    | ( CU          --> U                   at base -- dependances
      | H           --> (X,Y,STATE,OUTPUT) at base
      | (H,CU)      --> seulH               at base
      | (seulH,CU) --> X                    at base
      | U  --> X at CU
      | Y  --> X at seulH
      | OUTPUT --> Y      at H
      | OUTPUT --> STATE at H
      | X      --> STATE at H )
    | ( STATE  :  Xi = X at H                    -- actions
      | OUTPUT :  Y  = Xi at H ) )
    | ( X = U at CU                              -- flots
      | X = (Y-1) at seulH ) ) )

end MUX_1
```

Figure D.3 : *Le programme* MUX_1 *simplifié et ordonnancé*

```
metagc MUX_2

    ( ( clock(Y) = clock(B) = H     -- horloges
      | K = tt(B)
      | clock(U) = K )
    | ( B = (Y =< 0) at H ) )       -- flots

end MUX_2
```

Figure D.4 : *Le programme* MUX_2

```
metagc MUX

  act STATE, OUTPUT
  var Xi

    ( ( clock(X) = clock(Y) = clock(B) = H    -- horloges
      | clock(STATE) = clock(OUTPUT) = H
      | K = tt(B)
      | clock(U) = K
      | seulH = H whennot K )
    | ( OUTPUT --> STATE at H )               -- dependances
    | ( STATE  :  Xi = X at H                 -- actions
      | OUTPUT :  Y  = Xi at H )
    | ( X = U         at K                    -- flots
      | X = (Y-1)     at seulH
      | B = (Y =< 0) at H ) )

end MUX
```

Figure D.5 : *Le programme* MUX

```
metagc MUX

  act STATE, OUTPUT
  var Xi

    ( ( clock(X) = clock(Y) = clock(B) = H   -- horloges
      | clock(STATE) = clock(OUTPUT) = H
      | K = tt(B)
      | clock(U) = K
      | seulH = H whennot K )
    | ( H --> (X,Y,B,K,STATE,OUTPUT) at base -- dependances
      | K --> U at base
      | (H,K) --> seulH at base
      | Y --> B at H
      | B --> K at H
      | (K,seulH) --> X at base
      | U  --> X at K
      | Y  --> X at seulH
      | OUTPUT --> Y     at H
      | OUTPUT --> STATE at H
      | X      --> STATE at H )
    | ( STATE  :  Xi = X at H                -- actions
      | OUTPUT :  Y  = Xi at H )
    | ( X = U        at K                    -- flots
      | X = (Y-1)    at seulH
      | B = (Y =< 0) at H ) )

end MUX
```

Figure D.6 : *Le programme* MUX *ordonnancé*

```
metagc MUX_ASYNCH

  act STATE, OUTPUT
  var Xi

    ( ( clock(X) = clock(Y) = clock(B) = H    -- horloges
      | clock(STATE) = clock(OUTPUT) = H
      | clock(U) = tt(B) )
    | ( H --> (X,Y,B,STATE,OUTPUT) at H       -- dependances
      | Y --> B at H
      | (B,C) --> X at H
      | U  --> X at tt(B)
      | Y  --> X at tt(C)
      | OUTPUT --> Y     at H
      | OUTPUT --> STATE at H
      | X      --> STATE at H )
    | ( STATE  :  Xi = X at H                 -- actions
      | OUTPUT :  Y  = Xi at H )
    | ( X = U         at tt(B)                -- flots
      | X = (Y-1)     at tt(C)
      | B = (Y =< 0) at H
      | C = not B at H ) )

end MUX_ASYNCH
```

Figure D.7 : *Le programme* MUX_ASYNCH

Index

Prédéfinis

Symboles

Séparateurs

Lexique

Unité de Recherche INRIA Rennes
IRISA, Campus Universitaire de Beaulieu 35042 RENNES Cedex (France)

Unité de Recherche INRIA Lorraine Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 VILLERS LES NANCY Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 GRENOBLE Cedex (France)
Unité de Recherche INRIA Rocquencourt Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)
Unité de Recherche INRIA Sophia Antipolis 2004, route des Lucioles - B.P. 93 - 06902 SOPHIA ANTIPOLIS Cedex (France)

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399